Advanced R programming

Author: Hadley Wickham

Contents

Chapter 1

Introduction

I have spent over 10 years programming in R, spending a lot of time trying to figure out how the language works. Not everyone has the luxury of spending years to understand a programming language, so this book is my attempt to help you to become an effective R programmer as quickly as possible. By reading this book, you will avoid the many mistakes that I made and the many dead ends that I got stuck in, and quickly navigate your way to useful tools and techniques. Although R has its frustrating quirks, I truly believe that at its heart lies an elegant and beautiful language, well tailored for data analysis and statistics. In this book, I'll do my best to reveal that language to you, helping you to understand powerful idioms that allow you to attack many types of problems.

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

- It is free, open source and easily available on every major platform: if you do your analysis in R, anyone else can replicate it for free.
- A massive set of packages for statistical modelling, machine learning, visualisation, data import and data manipulation. Whatever model or visualisation you're trying to do, the chances are that someone has tried to do it before and you can learn from their efforts.
- Researchers in statistics and machine learning will often publish an R package to accompany their research articles. That means you can use cutting edge research as soon as it's avaiable.
- Deep features of the language support data analysis. These included missing values, data frames, and subsetting.
- A fantastic community. It is easy to get help from experts on the R-help

mailing list[1], stackoverflow[2], or subject specific mailing lists like R-SIG-mixed-models[3] or ggplot2[4]. You can also connect with other R learners via twitter[5], linkedin[6], and throuh many local user groups[7].

- Powerful tools for communicating your results. R packages make it easy to produce html or pdf reports[8], or create interactive websites[9].

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

- An IDE[10] tailored to the needs of interactive data analysis and statistical programming.

- Powerful metaprogramming facilities. R is not just a programming language, but it is also an environment for interactive data analysis. Metaprogramming makes it possible to write succinct functions that are a little magical in their ability to reduce the amount of typing you need. R also provides an excellent environment for designing domain specific languages.

- R is 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 R code you'll see in the wild is not very elegant, fast or easy to understand. It's usually written in haste to solve a pressing problem, and has not been rewritten for clarity, elegance or performance.

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

- Metaprogramming is a double-edged sword, and 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 fails in unexpected ways.

[1]https://stat.ethz.ch/mailman/listinfo/r-help
[2]http://stackoverflow.com/questions/tagged/r
[3]https://stat.ethz.ch/mailman/listinfo/r-sig-mixed-models
[4]https://groups.google.com/forum/#!forum/ggplot2
[5]https://twitter.com/search?q=%23rstats
[6]http://www.linkedin.com/groups/R-Project-Statistical-Computing-77616
[7]http://blog.revolutionanalytics.com/local-r-groups.html
[8]http://yihui.name/knitr/
[9]http://www.rstudio.com/shiny/
[10]http://www.rstudio.com/ide/

- Inconsistency is rife across contributed packages and even within base R. The R APIs have evolved over 20 years, and you are confronted with that fact every time you use R. Having to memorise many special cases makes learning R tough.

- 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, and tools for parallel processing are not mature.

Personally, I think these challenges are great opportunities for experienced programmers to have a profound positive impact on 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 it. I hope this book will help more R users to become R programmers, and encourage programmers from other languages to contribute to R.

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 more 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 will need to have written a decent amount of code in either R or other programming languages. You should be familiar with how functions work in R, although you might not know all the details, and you should be somewhat familiar with the apply family of functions (like `apply()` and `lapply()`), although you may currently struggle to use them effectively.

What you will get out of this book

This book describes the skills that I think you need to be an advanced R programmer, producing reusable code that can be used in a wide variety of circumstances.

After reading this book, you will:

- Be familiar with the fundamentals of R, so that you can understand complex data types and simplify the operations performed 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 useful tool for data analysis. You'll be able to use many existing new tools, and have the knowledge to create your own new 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 languages for expressing important operations. You'll also understand the dangers of metaprogramming, why you should be careful about its use.
- Have a good intuition for what 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.

Meta-techniques

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

Reading source code is a tremendously useful technique because it exposes you to new ways of doing things. Over time you'll develop a sense of taste as an R programmer, and even if you find something your taste violently objects to, it's still helpful: emulate the things you like and avoid the things you don't like. I think the clarity of my code increased considerably once I started grading code in the classroom, and was exposed to a lot of code I couldn't make heads nor tails of! I think it's a great idea to start by reading the source code for the functions and packages that you use most frequently. Reading the source becomes even more important when you start using more esoteric parts of R; often the documentation will be lacking, and you'll need to figure out how a function works by reading the source and experimenting.

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 if you can't figure it out and need to get help from others: you can easily show what you tried, and when you learn the right answer, you'll be mentally prepared to update your world view. I often find that whenever I make the effort to explain a problem so that others can understand and help to solve it (the art of a reproducible example[11]), I figure out the solution myself.

[11] http://stackoverflow.com/questions/5963269

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 refer to resources from other programming languages. R has aspects of both functional and object-oriented (OO) programming languages, and learning how these aspects are expressed in R will help you translate your existing knowledge from other programming languages, and to 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[12] (SICP), by Harold Abelson and Gerald Jay Sussman, particularly helpful. It is a concise but deep book, and after reading it I felt for the first time that I could actually design my own object-oriented system. It was my first introduction to the generic function style of OO common in R, and it helped me understand its strengths and weaknesses. SICP also talks a lot about functional programming, and how to create functions that are simple in isolation, but powerful in combination.

To understand the tradeoffs that R has made compared to other programming languages, I found Concepts, Techniques and Models of Computer Programming[13] 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 while the current implementation in R is not very efficient, that 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[14], by Andrew Hunt and David Thomas. This book is program language agnostic, and provides great advice for how to be a better programmer.

Getting help

Currently, there are two main venues to get help when you are stuck and can't figure out what's causing the problem: stackoverflow[15] and the R-help mailing list. You can get fantastic help in both venues, but they do have their own culture and expectations. It's usually a good idea to spend a little time lurking, and learning about community expectations before your first post.

Some good general advice:

[12] http://mitpress.mit.edu/sicp/full-text/book/book.html
[13] http://amzn.com/0262220695?tag=devtools-20
[14] http://amzn.com/020161622X?tag=devtools-20
[15] http://stackoverflow.com

- Make sure you have the latest version of R, and the package (or packages) you are having problems with. It may be that your problem is the result of a bug that has been fixed recently.
- Spend some time creating a reproducible example[16]. This is often a useful process in its own right, because in the course of making the problem reproducible you figure out what's causing the problem.
- Look for related problems before posting. If someone has already asked the question and been answered, it's much faster for everyone if you use the existing answer.

Acknowledgements

I would like to thank the tireless contributors to R-help and, more recently, stackoverflow[17]. 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 giving deeply of their time and correcting my countless misunderstandings.

This book was written in the open[18], and chapters were advertised on twitter[19] when complete. It is truly a community effort: many people read the 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.

(Before final version, remember to use `git shortlog` to list all contributors)

Colophon

This book was written in Rmarkdown[20] inside Rstudio[21]. knitr[22] and pandoc[23] converted the raw Rmarkdown to html and pdf. The website[24] was made with jekyll[25], styled with bootstrap[26], and published to amazon's S3[27] with

[16]http://stackoverflow.com/questions/5963269
[17]http://stackoverflow.com/questions/tagged/r
[18]https://github.com/hadley/adv-r/
[19]https://twitter.com/hadleywickham
[20]http://www.rstudio.com/ide/docs/authoring/using_markdown
[21]http://www.rstudio.com/ide/
[22]http://yihui.name/knitr/
[23]http://johnmacfarlane.net/pandoc/
[24]http://adv-r.had.co.nz
[25]http://jekyllrb.com/
[26]http://getbootstrap.com/
[27]http://aws.amazon.com/s3/

s3_website[28]. The complete source is available from github[29].

The code font is inconsolata[30].

[28] https://github.com/laurilehmijoki/s3_website
[29] https://github.com/hadley/adv-r
[30] http://levien.com/type/myfonts/inconsolata.html

Part I

Foundations

Chapter 2

Data structures

This chapter summarises the most important data structures in base R. I assume you've used many (if not all) of them before, but you may not have thought deeply about how they are interrelated. It is a brief overview: the goal is not to discuss individual types in depth, but to show how they fit together as a whole. I also expect that you'll read the documentation if you want more details on any of the specific functions used in the chapter.

R's base data structures are summarised in the table below, organised by their dimensionality and whether they're homogeneous (all contents must be of the same type) or heterogeneous (the contents can be of different types):

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

Note that R has no scalar, or 0-dimensional, types. All scalars (single numbers or strings) are length-one vectors.

Almost all other objects in R are built upon these foundations, and in the OO field guide[1] you'll see how R's object oriented tools build on top of these basics. There are also a few types of more esoteric objects that I don't describe here, but you'll learn about in depth in other parts of the book:

- functions[2], including closures and promises

[1] OO-essentials.html
[2] Functions.html

- environments[3]
- names/symbols, calls and expression objects, for metaprogramming[4]

When trying to understand the structure of an arbitrary object in R your most important tool is `str()`, short for structure: it gives a compact human readable description of any R data structure.

The chapter starts by describing R's 1d structures (atomic vectors and lists), then detours to discuss attributes (R's flexible metadata specification) and factors, before returning to discuss high-d structures (matrices, arrays and data frames).

Quiz

Take this short quiz to determine if you need to read this chapter. If you can bring the answers to mind quickly, you can comfortably skip this chapter.

- What are the three properties of a vector? (apart from its contents)
- What are the four common types of atomic vector? What are the two rarer types?
- What are attributes? How do you get and set them?
- How is a list different to a vector?
- How is a matrix different to a data frame?
- Can you have a list that is a matrix?
- Can a data frame have a column that is a list?

Vectors

The basic data structure in R is the vector, which comes in two basic flavours: atomic vectors and lists. They differ in their content: the contents of an atomic vector must all be the same type, the contents of a list can have different types. Atomic is so named because it forms the "atoms" of R's data structures. As well as their content, vectors have three properties: `typeof()` (what it is), `length()` (how long it is) and `attributes()` (additional arbitrary metadata). The most common attribute is `names()`.

Each type of vector comes with an `as.*` coercion function and an `is.*` testing function. But beware `is.vector()`: for historical reasons it returns `TRUE` only if the object is a vector with no attributes apart from names. Use `is.atomic(x) || is.list(x)` to test if an object is actually a vector.

[3] Environments.html
[4] metaprogramming.html

Atomic vectors

Atomic vectors can be logical, integer, double (often called numeric), or character, or less commonly complex or raw. Atomic vectors are usually created with `c()`, short for combine:

```
numeric <- c(1, 2.5, 4.5)
# Note the L suffix distinguishes doubles from integers
integer <- c(1L, 6L, 10L)
# Use T and F or TRUE and FALSE to create logical vectors
logical <- c(T, FALSE, TRUE, FALSE)
character <- c("these are", "some strings")
```

Atomic vectors are flat, and nesting `c()` just creates a flat vector:

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

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

Types and tests

Given a vector, you can determine what type it is with `typeof()`, or with a specific test: `is.character()`, `is.double()`, `is.integer()`, `is.logical()`, or, more generally, `is.atomic()`.

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

```
integer <- c(1L, 6L, 10L)
typeof(integer)
#> [1] "integer"
is.integer(integer)
#> [1] TRUE
is.double(integer)
#> [1] FALSE
is.numeric(integer)
#> [1] TRUE
```

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

Coercion

An atomic vector can only be of one type, so when you attempt to combine different types they will be **coerced** into one type, picking the first present type from character, double, integer and logical.

```
c("a", 1)
#> [1] "a" "1"
```

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

```
as.numeric(c(F, F, T))
#> [1] 0 0 1
# Total number of TRUEs
sum(mtcars$cyl == 4)
#> [1] 11
# Proportion of TRUEs
mean(mtcars$cyl == 4)
#> [1] 0.3438
```

You can manually force one type of vector to another using a coercion function: `as.character()`, `as.double()`, `as.integer()`, `as.logical()`. Coercion often also happens automatically. Most mathematical functions (`+`, `log`, `abs`, etc.) will coerce to a double or integer, and most logical operations (`&`, `|`, `any`, etc) will coerce to a logical. You will usually get a warning message if the coercion might lose information. If confusion is likely, it's better to explicitly coerce.

Lists

Lists are different from atomic vectors in that they can contain any other type of vector, including lists. You construct them using `list()` instead of `c()`.

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

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

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

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

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

The `typeof()` a list is `list`, you can test for a list with `is.list()` and coerce to a list with `as.list()`.

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

```
is.list(mtcars)
#> [1] TRUE
names(mtcars)
#>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#> [11] "carb"
str(mtcars$mpg)
#>  num [1:32] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...

mod <- lm(mpg ~ wt, data = mtcars)
is.list(mod)
#> [1] TRUE
names(mod)
#>  [1] "coefficients" "residuals"     "effects"       "rank"
#>  [5] "fitted.values" "assign"        "qr"            "df.residual"
#>  [9] "xlevels"       "call"          "terms"         "model"
str(mod$qr)
#> List of 5
#>  $ qr   : num [1:32, 1:2] -5.657 0.177 0.177 0.177 0.177 ...
#>   ..- attr(*, "dimnames")=List of 2
#>   .. ..$ : chr [1:32] "Mazda RX4" "Mazda RX4 Wag" "Datsun 710" "Hornet 4 Drive" ...
#>   .. ..$ : chr [1:2] "(Intercept)" "wt"
#>   ..- attr(*, "assign")= int [1:2] 0 1
#>  $ qraux: num [1:2] 1.18 1.05
#>  $ pivot: int [1:2] 1 2
#>  $ tol  : num 1e-07
#>  $ rank : int 2
#>  - attr(*, "class")= chr "qr"
```

You can turn a list back into an atomic vector using `unlist()`: this uses the same implicit coercion rules as for `c()`.

Exercises

1. What are the six types of atomic vectors? How does a list differ from an atomic vector?

2. What makes `is.vector()` and `is.numeric()` fundamentally different to `is.list()` and `is.character()`?

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

   ```
   c(1, F)
   #> [1] 1 0
   c("a", 1)
   #> [1] "a" "1"
   ```

```
c(list(1), "a")
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] "a"
c(T, 1L)
#> [1] 1 1
```

4. Why is `1 == "1"` true? Why is `-1 < 0` true? Why is `"one" < 2` false?
5. Why is the default (and shortest) `NA` a logical vector? What's special about logical vectors?

Attributes

All objects can have arbitrary additional attributes. These can be thought of as a named list (with unique names). Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`.

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

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

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

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

```
y[1]
#> [1] 1
sum(y)
#> [1] 55
```

The exceptions are for the most common attributes:

- `names()`, character vector of element names
- `class()`, used to implement the S3 object system, described in the next section
- `dim()`, used to turn vectors into high-dimensional structures

You should always get and set these attributes with their accessor functions: use `names(x)`, `class(x)` and `dim(x)`, not `attr(x, "names")`, `attr(x, "class")`, and `attr(x, "dim")`.

Names

You can give a vector names in three ways:

- During creation: `x <- c(a = 1, b = 2, c = 3)`
- By modifying a vector in place: `x <- 1:3; names(x) <- c("a", "b", "c")`
- By creating a modified vector: `x <- setNames(1:3, c("a", "b", "c"))`

Names should be unique, because character subsetting (see subsetting[5]), the biggest reason to use names, will only return the first match.

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

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

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

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

Factors

A factor is a vector that can contain only predefined values, and is R's structure for dealing with qualitative data. A factor is not a basic vector, but is built on top of an integer vector using the S3 class system, as described in the OO field guide[6]. Factors have two key attributes: their `class()`, "factor", which controls their behaviour; and their `levels()`, the set of allowed values.

[5] Subsetting.html
[6] OO-essentials.html

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

# You can't use values not in levels
x[2] <- "c"
#> Warning:     ,  NA
x
#> [1] a    <NA> b    a
#> Levels: a b

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

While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings. Some string methods (like `gsub()` and `grepl()`) will coerce factors to strings, while others (like `nchar()`) will throw an error, and still others will use the underlying integer ids (like `c()`). For this reason, it's usually best to explicitly convert factors to strings when modifying their levels.

Factors are useful when you know the possible values a variable may take, even if you don't see them in the dataset. Using a factor instead of a character vector makes it obvious when some groups contain no observations:

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

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

Sometimes when you read a data frame from a file, a column you'd thought would be numeric is actually a factor, with the numeric values appearing in the levels. This is caused by a non-numeric value in that column, often from a non-standard missing value. To remedy this you have two options. The best option is to discover what caused the problem and fix it in your data loading code (often by using the `na.value` argument to `read.csv()`). If you can't do that, you'll need to first coerce the factor to a character vector and then to numeric:

```r
z <- factor(c(12, 1, 9))
# Oops, that's not right
as.numeric(z)
#> [1] 3 1 2
# Perfect :)
as.numeric(as.character(z))
#> [1] 12  1  9
```

Unfortunately, most data loading functions in R automatically convert character vectors to factors. This is suboptimal, because there's no way for those functions to know the set of all possible levels and their optimal order. Instead, use the argument `stringsAsFactors = FALSE` to suppress this behaviour, and then manually convert character vectors to factors using your knowledge of the data. A global option (`options(stringsAsFactors = FALSE`) is available to control this behaviour, but it's not recommended - it may have unexpected consequences when combined with other code (either from packages, or code that you're `source()`ing.) In early versions of R, there was a memory advantage to using factors; that is no longer the case.

Atomic vectors and lists are the building blocks for higher dimensional data structures. Atomic vectors extend to matrices and arrays, and lists are used to create data frames.

Exercises

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

  ```r
  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)

Matrices and arrays

A vector becomes a matrix (2d) or array (>2d) with the addition of a `dim()` attribute. They can be created using the `matrix()` and `array()` functions, or by using the replacement form of `dim()`:

```
a <- matrix(1:6, ncol = 3)
b <- array(1:12, c(2, 3, 2))

c <- 1:6
dim(c) <- c(3, 2)
c
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
dim(c) <- c(2, 3)
c
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

`length()` generalises to `nrow()` and `ncol()` for matrices, and `dim()` for arrays. `names()` generalises to `rownames()` and `colnames()` for matrices, and `dimnames()` for arrays.

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

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

```
#> , , B
#>
#>     a  b  c
#> one 7  9 11
#> two 8 10 12
```

`c()` generalises to `cbind()` and `rbind()` for matrices, and to `abind::abind()` for arrays.

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

Beware that there are a few different ways to create a 1d datastructure: you can have a vector, row vector, column vector, or a 1d array. They may print similarly, but will behave differently. As always, use `str()` to reveal the differences.

```
str(list(
  vector = 1:3,
  col_vector = matrix(1:3, ncol = 1),
  row_vector = matrix(1:3, nrow = 1),
  array = array(1:3, 3)
))
#> List of 4
#>  $ vector    : int [1:3] 1 2 3
#>  $ col_vector: int [1:3, 1] 1 2 3
#>  $ row_vector: int [1, 1:3] 1 2 3
#>  $ array     : int [1:3(1d)] 1 2 3
```

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

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

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

Data frames

A data frame is the most common way of storing data in R, and if used systematically[7] make data analysis easier. Under the hood, a data frame is a list of equal-length vectors. This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list. This means that a data frame has `names()`, `colnames()` and `rownames()`, although `names()` and `colnames()` are the same thing. The `length()` of a data frame is the length of the underlying list and so is the same as `ncol()`, `nrow()` gives the number of rows.

As described in subsetting[8], you can subset a data frame like a 1d structure (where it behaves like a list), or a 2d structure (where it behaves like a matrix).

Creation

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

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

Beware the default behaviour of `data.frame()` to convert strings into factors. Use `stringAsFactors = FALSE` to suppress this behaviour:

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

Testing and coercion

Because data frame is an S3 class, its type reflects the underlying vector used to build it: `list`. Instead you can look at its `class()` or test for a data frame with `is.data.frame()`:

[7] http://vita.had.co.nz/papers/tidy-data.pdf
[8] subsetting.html

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

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

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

Combining data frames

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

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

When combining column-wise, only the number of rows needs to match, and rownames are ignored. When combining row-wise, the column names must match. If you want to combine data frames that may not have all the same variables, use `plyr::rbind.fill()`

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

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

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

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

Special columns

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

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

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

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

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

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

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

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

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

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

Chapter 3

Subsetting

R's subsetting operators are powerful and fast, and mastering them allows you to express complex operations. Subsetting allows you to express common data manipulation operations very succinctly, in a way few other languages can match. 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.

Subsetting is hard to learn because you need to master a number of interrelated concepts:

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

This chapter starts by introducing you to subsetting atomic vectors with `[`, and then gradually extends your knowledge, first to more complicated data types (like arrays and lists), and then to the other subsetting operators. You'll then learn how subsetting and assignment can be combined, and finally, you'll see a large number of useful applications.

Data types

It's easiest to understand how subsetting works for atomic vectors, and then learn how it generalises to higher dimensions and other more complicated objects. We'll start by exploring the use of `[`, the most commonly used operator. The next section will discuss `[[` and `$`, the two other main subsetting operators.

Atomic vectors

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

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

NB: the number after the decimal point gives the original position in the vector.

There are five ways of subsetting `x`:

- with **positive integers**, which return elements at the specified positions.

  ```r
  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
  ```

- with **negative integers**, which omit elements at the specified positions

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

  It's an error to mix positive and negative integers in a single subset:

  ```r
  x[c(-1, 2)]
  #> Error:
  ```

- with a **logical vector**, which selects elements where the corresponding logical value is `TRUE`. This is probably the most useful type of subsetting, because you will usually generate the logical vector with another expression.

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

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

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

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

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

- with **nothing**, which returns the original vector unchanged. This is not useful in 1d but it's very useful in 2d. It can also be useful in conjunction with assignment.

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

- with **zero**, which returns a zero-length vector. This is not something you'd 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 subset with:

- a **character vector**, which returns 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
  ```

Lists

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

Matrices and arrays

You can subset higher-dimensional structures in three ways: with multiple vectors, with a single vector, or 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 now becomes useful because you use it when you want to return all the rows or all the columns.

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

By default, `[` will simplify the results to the lowest possible dimensionality. See simplifying vs. preserving subsetting for how to avoid this.

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

```
(vals <- outer(1:5, 1:5, FUN = "paste", sep = ","))
#>      [,1]  [,2]  [,3]  [,4]  [,5]
#> [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
#> [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
#> [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
#> [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
#> [5,] "5,1" "5,2" "5,3" "5,4" "5,5"
vals[c(4, 15)]
#> [1] "4,1" "5,3"
```

You can also subset higher-dimensional data structures with an integer matrix (or, if named, a character matrix). Each row in the matrix specifies the location of a value, with each column corresponding to a dimension in the array being subsetted. The result is a vector of values:

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

Data frames

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

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

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

# There are two ways to select columns from a data frame
# Like a list:
df[c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
# Like a matrix
df[, c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
```

```
# There's an important difference if you select a simple 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
```

S3 objects

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

S4 objects

There are also two additional subsetting operators that are needed for S4 objects: `@` (equivalent to `$`), and `slot()` (equivalent to `[[`). `@` is more restrictive than `$` in that it will return an error if the slot does not exist. These are described in more detail in the OO field guide[1].

Exercises

- 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, ]
  ```

- Why does `x <- 1:5; x[NA]` yield five missing values? Hint: why is it different to `x[NA_real_]`?

- 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)]
  ```

- Why does `mtcars[1:20]` return a error? How does it differ from the similar `mtcars[1:20, ]`?

[1] OO-essentials.html

- Implement a function that extracts the diagonal entries from a matrix (it should behave like `diag(x)` where `x` is a matrix).
- What does `df[is.na(df)] <- 0` do? How does it work?

Subsetting operators

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

You need `[[` when working with lists. This is because `[` only returns a list - it never gives you the contents of the list. To do that, use `[[`:

> "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[2]

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

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

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

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

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

[2] http://twitter.com/#!/RLangTip/status/118339256388304896

Simplifying vs. preserving subsetting

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

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

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

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

- **atomic vector**: removes names

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

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

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

- **factor**: drops any unused levels

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

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

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

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

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

`$`

`$` is a shorthand operator, where `x$y` is equivalent to `x[["y", exact = FALSE]]`. It's commonly used to access columns of a dataframe, e.g. `mtcars$cyl`, `diamonds$carat`.

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

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

```
mtcars$var
#> NULL

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

There's one important different between `$` and `[[` - `$` does partial matching:

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

If you want to avoid this behaviour you can set `options(warnPartialMatchDollar = TRUE)` - but beware that this is a global option and may affect behaviour in other code you have loaded (e.g. packages).

Missing/out of bounds indices

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

```
x <- 1:3
str(x[4])
#>  int NA
str(x[NA_real_])
#>  int NA
str(x[NULL])
#>  int(0)
```

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

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

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

Exercises

- 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)`)

Subsetting and assignment

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

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

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

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

# You can't combine integer indices with NA
x[c(1, NA)] <- c(1, 2)
#> Error:      NA
# 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
```

Indexing with a blank can be useful in conjunction with assignment because it will preserve the original object class and structure. Compare the following two expressions. In the first, `mtcars` will remain as a dataframe. In the second, `mtcars` will become a list.

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

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

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

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

Applications

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

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"

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

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

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 <- sample(3, 5, rep = T)

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

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

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

# Using match
id <- match(grades, info$grade)
info[id, ]
#>     grade      desc  fail
#> 1       1      Poor  TRUE
#> 2       2      Good FALSE
#> 3       3 Excellent FALSE
#> 1.1     1      Poor  TRUE
#> 2.1     2      Good FALSE

# Using rownames
rownames(info) <- info$grade
```

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

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

Random samples/bootstrap (integer subsetting)

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

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

# Randomly reorder
df[sample(nrow(df)), ]
#>   x y z
#> 3 2 4 c
#> 4 2 3 d
#> 6 3 1 f
#> 2 1 5 b
#> 1 1 6 a
#> 5 3 2 e
# Select 3 random rows
df[sample(nrow(df), 3), ]
#>   x y z
#> 2 1 5 b
#> 1 1 6 a
#> 5 3 2 e
# Select 10 bootstrap samples
df[sample(nrow(df), 10, rep = T), ]
#>     x y z
#> 5   3 2 e
#> 3   2 4 c
#> 5.1 3 2 e
#> 2   1 5 b
#> 1   1 6 a
#> 6   3 1 f
#> 3.1 2 4 c
#> 2.1 1 5 b
#> 1.1 1 6 a
#> 3.2 2 4 c
```

The arguments of `sample()` control the number of samples to extract, and whether or not sampling with replacement is done.

Ordering (integer subsetting)

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

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

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
#> 6 f 1 3
#> 5 e 2 3
#> 3 c 4 2
#> 1 a 6 1
#> 2 b 5 1
#> 4 d 3 2

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

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

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
```

Removing columns from data frame (character subsetting)

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

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

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

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

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

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

Selecting rows based on a condition (logical subsetting)

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

```
mtcars[mtcars$cyl == 4, ]
#>                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
#> Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
#> Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#> Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
mtcars[mtcars$cyl == 4 & mtcars$gear == 4, ]
#>                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

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

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

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

`subset()` is a specialised shorthand function for subsetting data frames, and saves some typing because you don't need to repeat the name of the data frame. You'll learn how it works in Computing on the language[3].

```
subset(mtcars, cyl == 4)
#>                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
#> Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
#> Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#> Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
subset(mtcars, cyl == 4 & gear == 4)
#>                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

[3] Computing-on-the-language.html

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:

```r
x <- sample(10) < 4
which(x)
#> [1] 2 5 6

unwhich <- function(x, n) {
  out <- rep_len(FALSE, n)
  out[x] <- TRUE
  out
}
unwhich(which(x), 10)
#>  [1] FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE 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. Also beware that `x[-which(y)]` is **not** equivalent to `x[!y]`: if `y` is all FALSE, `which(y)` will be `integer(0)` and `-integer(0)` is still `integer(0)`, so you'll get no values, instead of all values. In general, avoid switching from logical to integer subsetting unless you want, for example, the first or last `TRUE` value.

Exercises

1. How would you take a random sample from the columns of a data frame? (This is an important technique in random forests). Can you simultaneously sample the rows and columns in one step?

2. How would you select a random contiguous sample of m rows from a data frame containing n rows?

Chapter 4

Vocabulary

An important part of being a competent R programmer is having a good working vocabulary. Below, I have listed the functions that I believe constitute such a vocabulary. I don't expect you to be intimately familiar with the details of every function, but you should at least be aware that they all exist.

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

The basics

```
# The first functions to learn
?
str

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

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

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

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

# Functions
function
missing
on.exit
return, invisible

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

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

# Making vectors
c
rep, rep_len
seq, seq_len, seq_along
rev
sample
choose, factorial, combn
```

```
(is/as).(character/numeric/logical/...)

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

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

Common data structures

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

# Character manipulation
grep, agrep
gsub
strsplit
chartr
nchar
tolower, toupper
substr
paste
library(stringr)

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

# Array manipulation
array
dim
```

```
dimnames
aperm
library(abind)
```

Statistics

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

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

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

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

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

Working with R

```
# Workspace
ls, exists, rm
getwd, setwd
q
```

```
source
install.packages, library, require

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

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

I/O

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

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

# Files and directories
dir
basename, dirname, tools::file_ext
```

```
file.path
path.expand, normalizePath
file.choose
file.copy, file.create, file.remove, file.rename, dir.create
file.exists, file.info
tempdir, tempfile
download.file, library(downloader)
```

Chapter 5

Functions

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

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

In this chapter you will learn:

- the three main components of a function.
- how scoping works, the process that looks up values from names.
- how everything that happens in R is a result of a function call, even if it doesn't look like it.
- the three ways of supplying arguments to a function, how to call a function given a list of arguments, and the impact of lazy evaluation.
- about two types of special functions: infix and replacement functions.
- what a function can return, and how invisible return values work.

Components of a function

All R functions have three parts:

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

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

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

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

The assignment forms of `body()`, `formals()`, and `environment()` can also be used to modify functions. This is a useful technique which we'll explore in more detail in computing on the language[1].

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

Primitive functions

There is one exception to the rule that functions have three components. Primitive functions, like `sum`, call C code directly with `.Primitive()` and contain no R code. Therefore their `formals()`, `body()` and `environment()` are all `NULL`:

[1] Computing-on-the-language.html

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

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

Exercises

1. What function allows you to tell if an object is a function? What function allows you to tell if a function is a primitive function or not?
2. Create a list of all primitive functions in R. (Hint: use `ls("package:base", all = TRUE)` to get a list of all objects in the base package, `get()` to retrieve an object given its name, and the answer to the question above.)
3. What are the three important components of a function?
4. When does printing a function not show what environment it was created in?

Lexical scoping

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

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

Understanding scoping allows you to:

- build tools by composing functions, as described in functional programming[2]

[2] Functional-programming.html

- overrule the usual evaluation rules and compute on the language[3]

R has two types of scoping: **lexical scoping**, implemented automatically at the language level, and **dynamic scoping**, used in select functions to save typing during interactive analysis. We describe lexical scoping here because it is intimately tied to function creation. Dynamic scoping is described in the context of computing on the language[4].

Lexical scoping looks up symbol values based on how functions were nested when they were created, not how they are nested when they are called. With lexical scoping, you can figure out where the value of each variable will be looked up only by looking at the definition of the function, you don't need to know anything about how the function is called.

The "lexical" in lexical scoping doesn't correspond to the usual English definition ("of or relating to words or the vocabulary of a language as distinguished from its grammar and construction") but comes from the computer science term "lexing", which is part of the process that converts code represented as text to meaningful pieces that the programming language understands. R's lexical scoping is lexical in this sense because you only need the definition of the functions, not how they are called.

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

- name masking
- functions vs. variables
- a fresh start
- dynamic lookup

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

Name masking

The following example illustrates the simplest principle, and you should have no problem predicting the output.

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

[3]Computing-on-the-language.html
[4]Computing-on-the-language.html

```
f()
rm(f)
```

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

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

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

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

The same rules apply to closures, functions created by other functions. Closures will be described in more detail in functional programming[5]; here we'll just look at how they interact with scoping. The following function, `j()`, returns a function. What do you think this function will return when we call it?

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

[5] Functional-programming.html

```
k <- j(1)
k()
rm(j, k)
```

This seems a little magical (how does R know what the value of `y` is after the function has been called), but it works because `k` keeps around the environment in which it was defined, which includes the value of `y`. Environments[6] gives some pointers on how you can dive in and figure out what values are stored in the environment associated with each function.

Functions vs. variables

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

```
l <- function(x) x + 1
m <- function() {
  l <- function(x) x * 2
  l(10)
}
m()
#> [1] 20
rm(l, m)
```

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

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

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

[6] Environments.html

A fresh start

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

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

You might be surprised that it returns the same value, `1`, every time. This is because every time a function is called, a new environment is created to host execution. A function has no way to tell what happened the last time it was run; each invocation is completely independent. (We'll see some ways to get around this in functional programming[7].)

Dynamic lookup

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

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

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

You generally want to avoid this behaviour because it means the function is no longer self-contained. This is a common error - if you make a spelling mistake in

[7] Functional-programming.html

your code, you won't get an error when you create the function, and you might not even get one when you run the function, depending on what variables are defined in the global environment.

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

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

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

```
environment(f) <- emptyenv()
f()
#> Error:   "+"
```

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

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

```
"(" <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(100, (1 + 2))
#>   [1] 3 3 3 3 3 3 3 3 3 4 3 3 4 3 3 3 3 3 4 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3
#>  [36] 4 3 3 3 3 4 3 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 4 3 4 3 3 3 3 3 3 3 3 3 3
#>  [71] 3 4 3 3 3 3 3 4 3 3 3 3 3 3 3 3 3 3 4 3 3 3 3 3 4 3 3 4 3 3
rm("(")
```

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

Exercises

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

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

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

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

   ```
   f <- function(x) {
     f <- function(x) {
       f <- function(x) {
         x ^ 2
       }
       f(x) + 1
     }
     f(x) * 2
   }
   f(10)
   ```

Every operation is a function call

> To understand computations in R, two slogans are helpful:
>
> - Everything that exists is an object.
> - Everything that happens is a function call.
>
> — John Chambers

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

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

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

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

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

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

It is possible to override the definitions of these special functions, but this is almost certainly a bad idea. However, it can occasionally allow you to do something that would have otherwise been impossible. For example, this feature makes it possible for the `dplyr` package to translate R expressions into SQL expressions. The Domain specific languages[8] chapter discusses using this idea to create domain specific languages that allow you to concisely express new concepts using existing R constructs.

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

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

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

[8] dsl.html

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

Note the difference between `` `+` `` and `"+"`. The first one is the value of the object called `+`, and the second is a string containing the character `+`. The second version works because `lapply` can be given the name of a function instead of the function itself: if you read the source of `lapply()`, you'll see the first line uses `match.fun()` to find functions given their names.

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

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

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

That everything in R is represented as a function call is important to know for computing on the language[9].

Function arguments

It's useful to distinguish between the formal arguments and the actual arguments to a function. The formal arguments are a property of the function, whereas the actual or calling arguments vary each time you call the function. This section discusses how calling arguments are mapped to formal arguments, how you can call a function given a list of arguments, how default arguments work and the impact of lazy evaluation.

Calling functions

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

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

[9] Computing-on-the-language.html

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

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

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

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

These are good calls:

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

This is probably overkill:

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

And these are just confusing:

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

Calling a function given a list of arguments

Suppose you had a list of function arguments:

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

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

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

Default and missing arguments

Function arguments in R can have default values.

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

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

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

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

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

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

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

Lazy evaluation

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

```
f <- function(x) {
  10
}
system.time(f(Sys.sleep(10)))
#>
#>     0     0     0
```

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

```
f <- function(x) {
  force(x)
  10
}
system.time(f(Sys.sleep(10)))
#>
#>   0.060   0.056 10.000
```

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

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

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

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

This code is exactly equivalent to

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

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

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

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

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

# ls() evaluated in global environment:
f(ls())
#>  [1] "add"       "adders"    "adders2"   "args"      "collapse" "f"
#>  [7] "g"         "h"         "i"         "in_path"   "x"         "y"
```

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

- an expression giving the delayed computation, which can be accessed with `substitute` (see controlling evaluation[10] for more details)
- the environment where the expression was created and where it should be evaluated

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

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

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

}
```

We could implement "&&" ourselves:

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

[10] Computing-on-the-language.html#Non-standard-evaluation-in-subset

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

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

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

```
if (is.null(a)) stop("a is null")
#> Error: a is null
```

You could write:

```
!is.null(a) || stop("a is null")
#> Error: a is null
```

`...`

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

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

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

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

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

To capture `...` in a form that is easier to work with, you can use `list(...)`. (See Computing on the language[11] for other ways to capture `...` without evaluating the arguments).

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

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

```
sum(1, 2, na.mr = TRUE)
#> [1] 4
```

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

Exercises

1. Clarify the following list of odd function calls:

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

Special calls

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

[11]Computing-on-the-language.html

Infix functions

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

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

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

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

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

Or indeed between

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

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

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

"a" % % "b"
#> [1] "a b"
```

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

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

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

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

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

Replacement functions

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

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

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

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

```
library(pryr)
x <- 1:10
address(x)
#> [1] "0x137a4bd8"
second(x) <- 6L
address(x)
#> [1] "0x1332a9c0"
```

Built in functions that are implemented using `.Primitive` will modify in place

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

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

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

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

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

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

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

This means you can't do things like:

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

because that gets turned into the invalid code:

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

It's often useful to combine replacement and subsetting, and this works out of the box:

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

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

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

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

Exercises

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

Return values

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

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

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

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

  # complicated processing here
}
```

Functions can return only a single value, but this is not a limitation in practice because you can always return a list containing any number of objects.

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

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

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

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

This is notably different to languages like Java where you can modify the inputs to a function. This copy-on-modify behaviour has important performance consequences which are discussed in depth in profiling[12]. (Note that the performance consequences are a result of R's implementation of copy-on-modify semantics, they are not true in general. Clojure is a new language that makes extensive use of copy-on-modify semantics with limited performance consequences.)

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

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

It's generally a good idea to minimise the use of side effects, and where possible separate functions into pure and impure, isolating side effects to the smallest possible location. Pure functions are easier to test (because all you need to worry about are the input values and the output), and are less likely to work differently on different versions of R or on different platforms. For example, this is one of the motivating principles of ggplot2: most operations work on an object that represents a plot, and only the final `print` or `plot` call has the side effect of actually drawing the plot.

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

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

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

[12] Profiling.html

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

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

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

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

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

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

because that is parsed as:

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

`on.exit()`

Another more casual way of cleaning up is the `on.exit` function, which is called when the function terminates. It's not as fine grained as `tryCatch`, but it's a bit less typing.

```
in_dir <- function(path, code) {
  cur_dir <- getwd()
  on.exit(setwd(cur_dir))

  force(code)
}
```

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

Exercises

1. 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).

Chapter 6

OO field guide

Introduction

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

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

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

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

- **S4** works similarly to S3, but is more formal. There are two major differences to S3. S4 has formal class definitions, which describe the representation and inheritance for each class, and has special helper functions for defining generics and methods. S4 also has multiple dispatch, which means that generic functions can pick methods based on the class of any number of arguments, not just one.
- **Reference classes**, called RC for short, are quite different to S3 and S4. RC implements message passing OO, so methods belong to classes, not functions. `$` is used to separate objects and methods, so method calls look like `canvas$drawRect("blue")`. RC objects are also mutable: they don't use R's usual copy-on-modify semantics, but are modified in place. This makes them harder to reason about, but allows them to solve problems that are difficult to solve with S3 or S4.

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

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

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

Base types

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

Data structures[1] explained the most common base types (atomic vectors and lists), but base types also encompass functions, environments and other more

[1] data-structures.html

exotic objects likes names, calls and promises that you'll learn about later in the book. You can find out the the base type of an object with `typeof()`, or see a complete list of types in `?typeof()`. Be aware that the names of base types are not used consistently throughout R: the type and the corresponding "is" function may have different names:

```
# The type of a function is "closure", and the
# type of a primitive function is "builtin"
f <- function() {}
typeof(f)
#> [1] "closure"
is.function(f)
#> [1] TRUE

typeof(sum)
#> [1] "builtin"
is.primitive(sum)
#> [1] TRUE
```

You may have heard of `mode()` and `storage.mode()`. I recommend ignoring these functions because they just alias some of the names returned by `typeof()` for S compatibility. Their source code is simple, and I recommend reading it if you want to know exactly what they do.

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

S3

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

Recognising objects, generic functions and methods

Most objects that you encounter are S3 objects. You can check by testing that it's an object (`is.object(x)`), but it's not an S4 object (`!isS4(x)`). This check is automated by `pryr::otype()`, which provides an easy way to determine the OO system of an object:

```
library(pryr)

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

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

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

```
mean
#> function (x, ...)
#> UseMethod("mean")
#> <bytecode: 0xd31118>
#> <environment: namespace:base>
ftype(mean)
#> [1] "s3"      "generic"
```

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

The job of an S3 generic is to call an S3 method specialised for the given class. You can recognise S3 methods by their names, which look like `generic.class()`. For example, the Date method for the `mean()` generic is called `mean.Date()`, and the factor method for `print()` is called `print.factor()`. Most modern style guides discourage the use of `.` in function names because it makes them look like S3 methods. For example, is `t.test()` the `test` method for `t` objects? Similarly, the use of `.` in class names can also be confusing: is `print.data.frame()` the `print()` method for `data.frames`, or the `print.data()` method for `frames`? `pryr::ftype()` knows about these exceptions, so you can use it to figure out if a function is an S3 method or generic:

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

ftype(is.numeric)   # naming convention for testing and coercion
#> [1] "primitive" "generic"
ftype(as.numeric)   # there are no S3 generics for is and as
#> [1] "primitive" "generic"
```

You can see all the methods of a generic using the `methods()` function:

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

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

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

```
methods(class = "ts")
#>  [1] aggregate.ts     as.data.frame.ts cbind.ts*        cycle.ts*
#>  [5] diffinv.ts*      diff.ts          kernapply.ts*    lines.ts
#>  [9] monthplot.ts*    na.omit.ts*      Ops.ts*          plot.ts
#> [13] print.ts         time.ts*         [<-.ts*          [.ts*
#> [17] t.ts*            window<-.ts*     window.ts*       xyplot.ts*
#>
#>    Non-visible functions are asterisked
```

There's no way to list all S3 classes, because there's no central repository of them, as you'll learn in the following section.

Defining classes and creating objects

S3 is a simple and ad hoc system; it has no formal definition of a class. To make an object an instance of a class, you just take an existing base object and set the class attribute. You can do that during creation with `structure()`, or after the fact with `attr<-()`. However, if you're modifying an existing object, using `class<-()` will more clearly communicate your intent:

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

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

S3 objects are usually built on top of lists, or atomic vectors with attributes[2]. You can also turn functions into S3 objects. Other base types are either rarely seen in R, or have unusual semantics that don't work well with attributes.

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

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

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

Most S3 classes provide a constructor function:

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

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

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

```
# Create a linear model
mod <- lm(log(mpg) ~ log(disp), data = mtcars)
class(mod)
#> [1] "lm"
```

[2]data-structures.html#attributes

```
# Turn it into a table (?!)
class(mod) <- "table"
# But unsurprisingly this doesn't work very well
print(mod)
#>
#>
#>
#>
#>
#>
#>
#>
#>
#>
#>
#>
#>
#> -5.657e+00, 1.768e-01, 1.768e-01, 1.768e-01, 1.768e-01, 1.768e-01, 1.768e-01, 1.768e-
#>
#>
#>
#>
#>
#>
#>
#>
#>
#>
```

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

Creating new methods and generics

To add a new generic, create a function that calls `UseMethod()`. `UseMethod()` takes two arguments: the name of the generic function, and the argument to use for method dispatch. If you omit the second argument it will dispatch on the first argument to the function. You don't pass `UseMethod()` any of the arguments to the generic - it uses black magic to find them out for itself.

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

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

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

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

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

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

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

Method dispatch

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

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

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

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

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

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

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

Because methods are normal R functions, you can call them directly. However, this is just as dangerous as changing the class of an object, so you shouldn't do it: please don't point the loaded gun at your foot! (The only reason to call the method directly is that sometimes when you're writing OO code, not using someone else's, you can get considerable speed-ups by skipping regular method dispatch.)

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

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

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

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

Exercises

- Read the source code for `t()` and `t.test()` and confirm for yourself that `t.test()` is an S3 generic, and not an S3 method. What happens if you create an object with class `test` and call `t()` with it?

- What classes have a method for the `Math` group generic in base R? Read the source code. How do the methods work?

- R has two classes for representing date time data, `POSIXct` and `POSIXlt` which both inherit from `POSIXt`. Which generics have different behaviour for the two classes? Which generics share the same behaviour?

- Which base generic has the most methods defined for it?

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

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

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

  h("a")
  ```

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

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

length.function <- function(x) "function"

length(f)
length(g)
```

S4

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

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

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

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

- S4 system development in Bioconductor[3]
- John Chambers' Software for Data Analysis[4]
- Martin Morgan's answers to S4 questions on stackoverflow[5]

[3]http://www.bioconductor.org/help/course-materials/2010/AdvancedR/S4InBioconductor.pdf
[4]http://amzn.com/0387759352?tag=devtools-20
[5]http://stackoverflow.com/search?tab=votes&q=user%3a547331%20%5bs4%5d%20is%3aanswe

Recognising objects, generic functions and methods

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

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

```r
library(stats4)

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

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

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

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

You can determine the class of an S4 object with `class()` and test if an object inherits from a specific class with `is()`:

```r
class(fit)
#> [1] "mle"
#> attr(,"package")
#> [1] "stats4"
```

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

You can get a list of all S4 generics with `getGenerics()`, and a list of all S4 classes with `getClasses()`, but note that this list includes shim classes for S3 classes and base types. You can list all S4 methods with `showMethods()`, optionally restricting either by `generic` or by `class` (or both). It's also a good idea to supply `where = search()` to restrict to methods available from the global environment.

Defining classes and creating objects

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

An S4 class has three key properties:

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

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

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

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

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

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

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

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

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

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

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

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

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

Creating new methods and generics

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

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

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

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

Method dispatch

S4 method dispatch is the same as S3 dispatch if your classes only inherit from a single parent, and you only dispatch on one class. The main difference is how you set up default values: S4 uses the special class "ANY" to match any class and "missing" to match a missing argument. Like S3, S4 also has group generics, documented in `?S4groupGeneric`, and a way to call the "parent" method, `callNextMethod()`.

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

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

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

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

Exercises

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

RC

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

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

These properties make RC objects behave more like objects found in most other programming languages like Python, Ruby, Java and C#. Surprisingly, reference classes are implemented in R, not C: they are a special S4 class that wraps around an environment.

Defining classes and creating objects

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

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

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

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

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

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

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

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

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

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

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

An object is not very useful without some behaviour defined by **methods**. RC methods are associated with a class and can modify its fields in-place. In the following example, note that you access the value of fields with their name, and you modify them using `<<-`. (You'll learn more about `<<-` in Environments[6])

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

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

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

The final important argument to `setRefClass()` is `contains`, the name of a parent RC class to inherit behaviour from. The following example creates a new type of bank account that throws an error preventing the balance from going below 0.

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

[6] environments.html

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

Recognising objects and methods

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

Method dispatch

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

Exercises

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

Picking a system

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

If you are creating more complicated systems of interrelated objects, S4 may be more appropriate. A good example is the `Matrix` package by Douglas Bates and Martin Maechler. It is designed to efficiently store and compute with many different types of sparse matrix. As at version 1.0.12 it defines 110 classes and 18 generic functions. The package is well written and well commented, and the accompanying vignette (`vignette("Intro2Matrix", package = "Matrix")`) gives a good overview of the structure of the package. S4 is also used extensively by Bioconductor packages, which need to model complicated interrelationships of biological objects. Bioconductor provides many good resources[7] for learning S4. If you've mastered S3, S4 is relatively easy to pick up: the ideas are all the same, it is just more formal, more strict and more verbose.

If you've programmed in mainstream OO language, RC will seem very natural. But because they can introduce side-effects through mutable state, they are harder to understand. For example, when you usually call `f(a, b)` in R you can assume that `a` and `b` will not be modified; but if `a` and `b` are RC objects, they might be modified in the place. Generally, when using RC objects you want to minimise side effects as much as possible, and use them only where mutable state is absolutely required. The majority of functions should still be "functional[8]", and side effect free. This makes code easier to reason about and easier for other R programmers to understand.

[7] https://www.google.com/search?q=bioconductor+s4
[8] functional-programming.html

Chapter 7

Environments

Introduction

Understanding environment objects is an important next step of understanding scoping. This chapter will teach you:

- what an environment is and how to inspect and manipulate environments
- the four types of environment associated with a function
- how to work in an fresh environment outside of a function with `local()`
- four ways of binding names to values in an environment

The [[functions]] chapter focusses on the essence of how scoping works, where this chapter will focus more on the details and show you how you can implement the behaviour yourself. It also introduces some ideas that will be useful for [[computing on the language]].

This chapter uses many functions found in the `pryr` package to pry open the covers of R and look inside the messy details. Install `pryr` by running `devtools::install_github("pryr")`

Environment basics

What is an environment?

The job of an environment is to associate, or **bind**, a set of names to values. Environments are the data structures that power scoping. An **environment** is very similar to a list, with three important exceptions:

- Environments have reference semantics: R's usual copy on modify rules do not apply. Whenever you modify an environment, you modify every copy.

  In the following code chunk, we create a new environment, create a "copy" and then modify the original environment. The copy also changes. If you change `e` to a list (or any other R datastructure) `e` and `f` are independent.

  ```
  e <- new.env()
  f <- e

  e$a <- 10
  f$a
  #> [1] 10
  ```

- Environments have parents: if an object is not found in an environment, then R can look in its parent (and so on). There is only one exception: the **empty** environment does not have a parent.

  We use the family metaphor to refer to other environments: the grandparent of a environment would be the parent's parent, and the ancestors include all parent environments all the way up to the empty environment. It's rare to talk about the children of an environment because there are no back links: given an environment we have no way to find its children.

- Every object in an environment must have a name, and the names must be unique.

Technically, an environment is made up of a **frame**, a collection of named objects (like a list), and a reference to a parent environment.

As well as powering scoping, environments can also be useful data structures because unlike almost every other type of object in R, modification takes place without a copy. This is not something that you should use without thought: it will violate users' expectations about how R code works, but it can sometimes be critical for high performance code. However, since the addition of [[R5]], you're generally better off using reference classes instead of raw environments. Environments can also be used to simulate hashmaps common in other packages, because name lookup is implemented with a hash, which means that lookup is O(1). See the CRAN package hash for an example.

Manipulating and inspecting environments

You can create environments with `new.env()`, see their contents with `ls()`, and inspect their parent with `parent.env()`.

```
e <- new.env()
# the default parent provided by new.env() is environment from which it is called
parent.env(e)
#> <environment: R_GlobalEnv>
identical(e, globalenv())
#> [1] FALSE
ls(e)
#> character(0)
```

You can modify environments in the same way you modify lists:

```
ls(e)
#> [1] "a"
e$a <- 1
ls(e)
#> [1] "a"
e$a
#> [1] 1
```

By default `ls` only shows names that don't begin with `.`. Use `all.names = TRUE` (or `all` for short) to show all bindings in an environment:

```
e$.a <- 2
ls(e)
#> [1] "a"
ls(e, all = TRUE)
#> [1] "a"  ".a"
```

Another useful technique to view an environment is to coerce it to a list:

```
as.list(e)
#> $a
#> [1] 1
str(as.list(e))
#> List of 1
#>  $ a: num 1
str(as.list(e, all.names = TRUE))
#> List of 2
#>  $ a : num 1
#>  $ .a: num 2
```

You can extract elements of an environment using `$` or `[[`, or `get`. `$` and `[[` will only look in that environment, but `get` uses the regular scoping rules and will also look in the parent, if needed. `$` and `[[` will return `NULL` if the name is not found, while `get` returns an error.

```
e$b <- 2
e$b
#> [1] 2
e[["b"]]
#> [1] 2
get("b", e)
#> [1] 2
```

Deleting objects from environments works a little different to lists. In a list you can remove an entry by setting it to `NULL`. That doesn't work in environments, and instead you need to use `rm()`.

```
e <- new.env()

e$a <- 1
e$a <- NULL
ls(e)
#> [1] "a"

rm("a", envir = e)
ls(e)
#> character(0)
```

Generally, when you create your own environment, you want to manually set the parent environment to the empty environment. This ensures you don't accidentally inherit objects from somewhere else:

```
x <- 1
e1 <- new.env()
get("x", e1)
#> [1] 1

e2 <- new.env(parent = emptyenv())
get("x", e2)
#> Error:    'x'
```

You can determine if a binding exists in a environment with the `exists()` function, but note that the default is to follow the regular scoping rules and will also look in the parent environments. If you don't want this behavior, use `inherits = FALSE`:

```
exists("b", e)
#> [1] FALSE
exists("b", e, inherits = FALSE)
#> [1] FALSE
exists("a", e, inherits = FALSE)
#> [1] FALSE
```

Special environments

There are a few special environments that you can access directly:

- `globalenv()`: the user's workspace
- `baseenv()`: the environment of the base package
- `emptyenv()`: the ultimate ancestor of all environments, the only environment without a parent.

The most common environment is the global environment (`globalenv()`) which corresponds to the top-level workspace. The parent of the global environment is one of the packages you have loaded (the exact order will depend on which packages you have loaded in which order). The eventual parent will be the base environment, which is the environment of "base R" functionality, which has the empty environment as a parent.

`search()` lists all environments in between the global and base environments. This is called the search path, because any object in these environments can be found from the top-level interactive workspace. It contains an environment for each loaded package and for each object (environment, list or Rdata file) that you've `attach()`ed. It also contains a special environment called `Autoloads` which is used to save memory by only loading package objects (like big datasets) when needed. You can access the environments of any environment on the search list using `as.environment()`.

```
search()
#>  [1] ".GlobalEnv"                "package:parallel"
#>  [3] "package:dplyr"             "package:hflights"
#>  [5] "package:pryr"              "package:Rcpp"
#>  [7] "package:lattice"           "package:microbenchmark"
#>  [9] "package:inline"            "package:knitr"
#> [11] "package:devtools"          "package:stats"
#> [13] "package:graphics"          "package:grDevices"
#> [15] "package:utils"             "package:datasets"
#> [17] "package:methods"           "Autoloads"
#> [19] "package:base"
as.environment("package:stats")
#> <environment: package:stats>
#> attr(,"name")
#> [1] "package:stats"
#> attr(,"path")
#> [1] "/usr/lib/R/library/stats"
```

Where

We can apply our new knowledge of environments to create a helpful function called `where` that tells us the environment where a variable lives:

```
library(pryr)
where("where")
#> <environment: package:pryr>
#> attr(,"name")
#> [1] "package:pryr"
#> attr(,"path")
#> [1] "/home/heihei/R/x86_64-pc-linux-gnu-library/3.0/pryr"
where("mean")
#> <environment: base>
where("t.test")
#> <environment: package:stats>
#> attr(,"name")
#> [1] "package:stats"
#> attr(,"path")
#> [1] "/usr/lib/R/library/stats"
x <- 5
where("x")
#> <environment: 0x13024738>
```

`where()` obeys the regular rules of variable scoping, but instead of returning the value associated with a name, it returns the environment in which it was defined.

The definition of `where()` is fairly straightforward. It has two arguments; the name to look for (as a string), and the environment in which to start the search. (We'll learn later why `parent.frame()` is a good default.)

```
where
#> function (name, env = parent.frame())
#> {
#>     stopifnot(is.character(name), length(name) == 1)
#>     env <- to_env(env)
#>     if (identical(env, emptyenv())) {
#>         stop("Can't find ", name, call. = FALSE)
#>     }
#>     if (exists(name, env, inherits = FALSE)) {
#>         env
#>     }
#>     else {
#>         where(name, parent.env(env))
#>     }
#> }
#> <environment: namespace:pryr>
```

It's natural to work with environments recursively, so we'll see this style of function structure frequently. There are three main components:

- the base case (what happens when we've recursed down to the empty environment)
- a boolean that determines if we've found what we wanted, and
- the recursive statement that re-calls the function using the parent of the current environment.

If we remove all the details of where, and just keep the structure, we get a function that looks like this:

```
f <- function(..., env = parent.frame()) {
  if (identical(env, emptyenv())) {
    # base case
  }

  if (success) {
    # return value
  } else {
    # inspect parent
    f(..., env = parent.env(env))
  }
}
```

Note that to check if the environment is the same as the empty environment, we need to use `identical()`: this performs a whole object comparison, unlike the element-wise `==`.

It is also possible to write this function with a loop instead of with recursion. This might run slightly faster (because we eliminate some function calls), but I find it harder to understand what's going on. I include it because you might find it easier to see what's happening if you're less familiar with recursive functions.

```
is.emptyenv <- function(x) identical(x, emptyenv())

f2 <- function(..., env = parent.frame()) {
  while(!is.emptyenv(env)) {
    if (success) {
      # return value
      return()
    }
    # inspect parent
    env <- parent.env(env)
  }

  # base case
}
```

Exercises

- Using `parent.env()` and a loop (or a recursive function), verify that the ancestors of `globalenv()` include `baseenv()` and `emptyenv()`. Use the same basic idea to implement your own version of `search()`.

- Write your own version of `get()` using a function written in the style of `where()`.

- 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. (This function should be a equivalent to `match.fun()` extended to take a second argument). For an added challenge, also add an `inherits` argument which controls whether the function recurses down the parents or only looks in one environment.

- Write your own version of `exists(inherits = FALSE)` (Hint: use `ls()`). Write a recursive version that behaves like `inherits = TRUE`.

Function environments

Most of the time when you are working with environments, you will not create them directly, but they will be created as a consequence of working with functions. This section discuss the four types of environment associated with a function.

There are multiple environments associated with each function, and it's easy to get confused between them.

- the environment where the function was created
- the environment where the function lives
- the environment that's created every time a function is run
- the environment where a function is called from

The following sections will explain why each of these environments are important, how to access them, and how you might use them.

The environment where the function was created

When a function is created, it gains a reference to the environment where it was made. This is the parent, or enclosing, environment of the function used by lexical scoping. You can access this environment with the `environment()` function:

```
y <- 1
f <- function(x) x + y
environment(f)
#> <environment: R_GlobalEnv>

environment(plot)
#> <environment: namespace:graphics>
environment(t.test)
#> <environment: namespace:stats>
```

To make an equivalent function that is safer (it throws an error if the input isn't a function), more consistent (can take a function name as an argument not just a function), and more informative (better name), we'll create `funenv()`:

```
funenv <- function(f) {
  f <- match.fun(f)
  environment(f)
}
funenv("plot")
#> <environment: namespace:graphics>
funenv("t.test")
#> <environment: namespace:stats>
```

Unsurprisingly, the enclosing environment is particularly important for closures:

```
plus <- function(x) {
  function(y) x + y
}
plus_one <- plus(1)
plus_one(10)
#> [1] 11
plus_two <- plus(2)
plus_one(10)
```

```
#> [1] 11
environment(plus_one)
#> <environment: 0x106f1e788>
parent.env(environment(plus_one))
#> <environment: R_GlobalEnv>
environment(plus_two)
#> <environment: 0x106e39c98>
parent.env(environment(plus_two))
#> <environment: R_GlobalEnv>
environment(plus)
#> <environment: R_GlobalEnv>
str(as.list(environment(plus_one)))
#> List of 1
#>  $ x: num 1
str(as.list(environment(plus_two)))
#> List of 1
#>  $ x: num 2
```

It's also possible to modify the environment of a function, using the assignment form of `environment`. This is rarely useful, but we can use it to illustrate how fundamental scoping is to R. One complaint that people sometimes make about R is that the function `f` defined above really should generate an error, because there is no variable `y` defined inside of R. Well, we could fix that by manually modifying the environment of `f` so it can't find y inside the global environment:

```
f <- function(x) x + y
environment(f) <- emptyenv()
f(1)
#> Error:  "+"
```

But when we run it, we don't get the error we expect. Because R uses its scoping rules consistently for everything (including looking up functions), we get an error that `f` can't find the `+` function. (See the discussion in [[scoping]] for alternatives that actually work.)

The environment where the function lives

The environment of a function, and the environment where it lives might be different. In the example above, we changed the environment of `f` to be the `emptyenv()`, but it still lived in the `globalenv()`:

```
f <- function(x) x + y
funenv("f")
#> <environment: R_GlobalEnv>
```

```
where("f")
#> <environment: R_GlobalEnv>
environment(f) <- emptyenv()
funenv("f")
#> <environment: R_EmptyEnv>
where("f")
#> <environment: R_GlobalEnv>
```

The environment where the function lives determines how we find the function, the environment of the function determines how it finds values inside the function. This important distinction is what enables package [[namespaces]] to work.

For example, take `t.test()`:

```
funenv("t.test")
#> <environment: namespace:stats>
where("t.test")
#> <environment: package:stats>
#> attr(,"name")
#> [1] "package:stats"
#> attr(,"path")
#> [1] "/usr/lib/R/library/stats"
```

We find `t.test()` in the `package::stats` environment, but its parent (where it looks up values) is the `namespace::stats` environment. The *package* environment contains only functions and objects that should be visible to the user, but the *namespace* environment contains both internal and external functions. There are over 400 objects that a defined in the `stats` package but not available to the user:

```
length(ls(funenv("t.test")))
#> [1] 1088
length(ls(where("t.test")))
#> [1] 493
```

This mechanism makes it possible for packages to have internal objects that can be accessed by its functions, but not by external functions.

The environment created every time a function is run

Recall how function scoping works. What will the following function return the first time we run it? What about the second?

```
f <- function(x) {
  if (!exists("a", inherits = FALSE)) {
    message("Defining a")
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
f()
#> Defining a

#> [1] 1
```

You should recall that it returns the same value every time. This is because every time a function is called, a new environment is created to host execution. We can see this more easily by returning the environment inside the function: using `environment()` with no arguments returns the current environment (try running it at the top level). Each time you run the function a new function is created. But they all have the same parent environment, the environment where the function was created.

```
f <- function(x) {
  list(
    e = environment(),
    p = parent.env(environment())
  )
}
str(f())
#> List of 2
#>  $ e:<environment: 0x10528b5f0>
#>  $ p:<environment: R_GlobalEnv>
str(f())
#> List of 2
#>  $ e:<environment: 0x106aa7a70>
#>  $ p:<environment: R_GlobalEnv>
funenv("f")
#>  <environment: R_GlobalEnv>
```

The environment where the function was called

Look at the following code. What do you expect `g()` to return when the code is run?

```
f <- function() {
  x <- 10
  function() {
    x
  }
}
g <- f()
x <- 20
g()
```

The top-level `x` is a red herring: using the regular scoping rules, `g()` looks first where it is defined and finds the value of `x` is 10. However, it is still meaningful to ask what value `x` is associated in the environment where `g()` is called. `x` is 10 in the environment where `g()` is defined, but it is 20 in the environment from which `g()` is **called**.

We can access this environment using the confusingly named `parent.frame()`. This function returns the **environment** from which the function was called. We can use that to look up the value of names in the environment from which the funtion was called.

```
f2 <- function() {
  x <- 10
  function() {
    def <- get("x", environment())
    cll <- get("x", parent.frame())
    list(defined = def, called = cll)
  }
}
g2 <- f2()
x <- 20
str(g2())
#> List of 2
#>  $ defined: num 10
#>  $ called : num 20
```

In more complicated scenarios, there's not just one parent call, but a sequence of calls all the way back to the initiating function, called from the top-level. We can get a list of all calling environments using `sys.frames()`

```
x <- 0
y <- 10
f <- function(x) {
  x <- 1
  g(x)
}
g <- function(x) {
  x <- 2
  h(x)
}
h <- function(x) {
  x <- 3
  i(x)
}
i <- function(x) {
  x <- 4
  sys.frames()
}

es <- f()
sapply(es, function(e) get("x", e, inherits = TRUE))
# [1] 1 2 3 4
sapply(es, function(e) get("y", e, inherits = TRUE))
# [1] 10 10 10 10
```

There are two separate strands of parents when a function is called: the calling environments, and the enclosing environments. Each calling environment will also have a stack of enclosing environments. Note that while called function has both a stack of called environments and a stack of enclosing environments, an environment (or a function object) has only a stack of enclosing environments.

Looking up variables in the calling environment rather than in the defining argument is called **dynamic scoping**. Few languages implement dynamic scoping (Emacs Lisp is a notable exception[1]) because dynamic scoping makes it much harder to reason about how a function operates: not only do you need to know how it was defined, you also need to know in what context it was called. Dynamic scoping is primarily useful for developing functions that aid interactive data analysis, and is one of the topics discussed in [[controlling evaluation]].

Exercises

- 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. Can you list objects that the function will be able to access but the user of the function cannot?

[1] http://www.gnu.org/software/emacs/emacs-paper.html#SEC15

Explicit scoping with `local`

Sometimes it's useful to be able to create a new scope without embedding inside a function. The `local` function allows you to do exactly that - it can be useful if you need some temporary variables to make an operation easier to understand, but want to throw them away afterwards:

```
df <- local({
  x <- 1:10
  y <- runif(10)
  data.frame(x = x, y = y)
})
```

This is equivalent to:

```
df <- (function() {
  x <- 1:10
  y <- runif(10)
  data.frame(x = x, y = y)
})()
```

(If you're familiar with JavaScript you've probably seen this pattern before: it's the immediately invoked function expression (IIFE) used extensively by most JavaScript libraries to avoid polluting the global namespace.)

`local` has relatively limited uses (typically because most of the time scoping is best accomplished using R's regular function based rules) but it can be particularly useful in conjunction with `<<-`. You can use this if you want to make a private variable that's shared between two functions:

```
a <- 10
my_get <- NULL
my_set <- NULL
local({
  a <- 1
  my_get <<- function() a
  my_set <<- function(value) a <<- value
})
my_get()
#> [1] 1
my_set(20)
a
#> [1] 10
my_get()
#> [1] 20
```

However, it can be easier to see what's going on if you avoid the implicit environment and create and access it explicitly:

```
my_env <- new.env(parent = emptyenv())
my_env$a <- 1
my_get <- function() my_env$a
my_set <- function(value) my_env$a <- value
```

These techniques are useful if you want to store state in your package.

Assignment: binding names to values

Assignment is the act of binding (or rebinding) a name to a value in an environment. It is the counterpart to scoping, the set of rules that determines how to find the value associated with a name. Compared to most languages, R has extremely flexible tools for binding names to values. In fact, you can not only bind values to names, but you can also bind expressions (promises) or even functions, so that every time you access the value associated with a name, you get something different!

The remainder of this section will discuss the four main ways of binding names to values in R:

- With the regular behaviour, `name <- value`, the name is immediately associated with the value in the current environment. `assign("name", value)` works similarly, but allows assignment in any environment.
- The double arrow, `name <<- value`, assigns in a similar way to variable lookup, so that `i <<- i + 1` modifies the binding of the original `i`, which is not necessarily in the current environment.
- Lazy assignment, `delayedAssign("name", expression)`, binds an expression isn't evaluated until you look up the name.
- Active assignment, `makeActiveBinding("name", function)` binds the name to a function, so it is "active" and can return different a value each time the name is found.

Regular binding

You have probably used regular assignment in R thousands of times. Regular assignment immediately creates a binding between a name and a value in the current environment.

There are two types of names: syntactic and non-syntactic. Generally, syntactic names consist of letters, digits, `.` and `_`, and must start with a letter or `.` not followed by a number (so `.a` and `._` are syntactic but `.1` is not). There are also a number of reserved words (e.g. `TRUE`, `NULL`, `if`, `function`, see `make.names()`). A syntactic name can be used on the left hand side of `<-`:

```
a <- 1
._ <- 2
a_b <- 3
```

However, a name can actually be any sequence of characters; if it's non-syntactic you just need to do a little more work:

```
`a + b` <- 3
`:)` <- "smile"
`    ` <- "spaces"
ls()
#  [1] "    "   ":)"     "a + b"
```

`<-` creates a binding in the current environment. There are three techniques to create a binding in another environment:

- treating an environment like a list

  ```
  e <- new.env()
  e$a <- 1
  ```

- use `assign()`, which has three important arguments: the name, the value, and the environment in which to create the binding

  ```
  e <- new.env()
  assign("a", 1, envir = e)
  ```

- evaluate `<-` inside the environment. (More on this in [[evaluation]])

  ```
  e <- new.env()

  eval(quote(a <- 1), e)
  # alternatively, you can use the helper function evalq
  # evalq(x, e) is exactly equivalent to eval(quote(x), e)
  evalq(a <- 1, e)
  ```

I generally prefer to use the first form because it is so compact. However, you'll see all three forms in R code in the wild.

Constants

There's one extension to regular binding: constants. What are constants? They're variable whose values can not be changed; they can only be bound once, and never re-bound. We can simulate constants in R using `lockBinding`, or the infix `%<c-%` found in pryr:

```
x <- 10
lockBinding(as.name("x"), globalenv())
x <- 15
rm(x)

x %<c-% 20 #>
x <- 30
#> Error:       'x'
rm(x)
```

`lockBinding()` is used to prevent you from modifying objects inside packages:

```
assign("mean", function(x) sum(x) / length(x), env = baseenv())
#> Error:       'mean'
```

`<<-`

Another way to modify the binding between name and value is `<<-`. The regular assignment arrow, `<-`, always creates a variable in the current environmnt. The special 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() {
  g <- function() {
    x <<- 2
  }
  x <- 1
  g()
  x
}
f()
#> [1] 2
x
#> [1] 0
```

```
h <- function() {
  x <- 1
  x <<- 2
  x
}
h()
#> [1] 1
x
#> [1] 2
```

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.

`name <<- value` is equivalent to `assign("name", value, inherits = TRUE)`.

To give you more idea how this works, we could implement `<<-` ourselves. I'm going to call it `rebind`, and emphasise that it's normally used to modify an existing binding. We'll implement it with our recursive recipe for working with environments. For the base case, we'll throw an error (where `<<-` would assign in the global environment), which emphasises the rebinding nature of this function. Otherwise we check to see if the name is found in the current environment: if it is, we do the assignment there; if not, we recurse.

```
rebind <- function(name, value, env = parent.frame()) {
  if (identical(env, emptyenv())) {
    stop("Can't find ", name, call. = FALSE)
  }

  if (exists(name, envir = env, inherits = FALSE)) {
    assign(name, value, envir = env)
  } else {
    rebind(name, value, parent.env(env))
  }
}
rebind("a", 10)
a <- 5
rebind("a", 10)
a
#> [1] 10

f <- function() {
  g <- function() {
    rebind("x", 2)
  }
  x <- 1
  g()
  x
}
f()
#> [1] 2
```

We'll come back to this idea in depth, and see where it is useful in [[functional programming]].

Delayed bindings

Another special type of assignment is a delayed binding: rather than assigning the result of an expression immediately, it creates and stores a promise to evaluate the expression when needed (much like the default lazy evaluation of arguments in R functions). We can create delayed bindings with the special assignment operator `%<d-%`, provided by the pryr package.

```
library(pryr)
a %<d-% 1
system.time(a)
#>
#>    0     0     0
b %<d-% {Sys.sleep(1); 1}
system.time(b)
#>
#> 0.008 0.004 1.000
```

Note that we need to be careful with more complicated expressions because user created infix functions have the lowest possible precendence: `x %<d-% a + b` is interpreted as `(x %<d-% a) + b`, so we need to use parentheses ourselves:

```
x %<d-% (a + b)
a <- 5
b <- 5
a + b
#> [1] 10
```

`%<d-%` is a wrapper around the base `delayedAssign()` function, which you may need to use directly if you need more control. `delayedAssign()` has four parameters:

- `x`: a variable name given as a quoted string
- `value`: an unquoted expression to be assigned to x

- `eval.env`: the environment in which to evaluate the expression
- `assign.env`: the environment in which to create the binding

Writing `%<d-%` is straightforward, bearing in mind that `makeActiveBinding` uses non-standard evaluation to capture the representation of the second argument, so we need to use substitute to construct the call manually. Once you've read [[computing on the language]], you might want to read the source code and think about how it works.

One application of `delayedAssign` is `autoload`, a function that powers `library()`. `autoload` makes R behave as if the code and data in a package is loaded in memory, but it doesn't actually do any work until you call one of the functions or access a dataset. This is the way that data sets in most packages work - you can call (e.g.) `diamonds` after `library(ggplot2)` and it just works, but it isn't loaded into memory unless you actually use it.

Active bindings

You can create **active** bindings where the value is recomputed every time you access the name:

```
x %<a-% runif(1)
x
#> [1] 0.4428
x
#> [1] 0.3233
```

`%<a-%` is a wrapper for the base function `makeActiveBinding()`. You may want to use this function directly if you want more control. It has three arguments:

- `sym`: a variable name, represented as a name object or a string
- `fun`: a single argument function. Getting the value of `sym` calls `fun` with zero arguments, and setting the value of `sym` calls `fun` with one argument, the value.
- `env`: the environment in which to create the binding.

Exercises

- In `rebind()` it's unlikely that we want to assign in an ancestor of the global environment (i.e. a loaded package), so modify the function to avoid recursing past the global environment.

- Create a version of `assign()` that will only bind new names, never re-bind old names. Some programming languages only do this, and are known as single assignment[2] languages.

[2] http://en.wikipedia.org/wiki/Assignment_computer_science#Single_assignment

- Write an alternative to `<-` that never overrides an existing binding. This would be useful if you are running a test script multiple times and only want to generate the test data once.

- Implement `str` for environments, listing all bindings in the environment, and briefly describing their contents (you might want to use `str` recursively). Use `bindingIsActive()` to determine if a binding is active. Indicate if bindings are locked (see `bindingIsLocked()`). Show the expressions (not the results) for delayed bindings (see the help for `delayedAssign` for hints). Show the amount of memory the environment occupies using `object.size()`

- Write an assignment function that can do active, delayed and locked bindings. What might you call it? What arguments should it take? Can you guess which sort of assignment it should do based on the expression?

Chapter 8

Debugging, condition handling and defensive programming

What happens when something goes wrong with your R code? What do you do? What tools do you have to apply to the problem? This chapter will teach you how you to fix unanticipated problems (debugging), show you how functions can communicate problems and how you can take action based on those communications (condition handling), and teach you how to avoid common problems before they occur (defensive programming).

Debugging is the art and science of fixing unexpected problems in your code. In this section you'll learn tools and techniques help you get to the root cause of an error when you encounter it. You'll learn general strategies for debugging, and RStudio and R specific tools like `traceback()` and `browser()`.

Not all problems are unexpected. When writing a function, you can often anticipate potential problems (like a file not existing, or the wrong type of input). Communicating these problems back to the user is the job of **conditions**, which include errors, warnings and messages:

- Fatal errors are raised by `stop()` and force all execution to terminate. Errors are used when there is no way for a function to continue.
- Warnings are generated by `warning()` and are used to display potential problems, such as when some elements of a vectorised input are invalid, like `log(-1:2)`.
- Messages are generated by `message()` and are used to give informative output in a way that can easily be suppressed by the user

(`?suppressMessages()`). I often use messages to let the user know what value the function has chosen for an important missing argument.

Conditions are usually displayed prominently, in a bold font or coloured red depending on your R interface. You can tell them apart because errors always start with "Error" and warnings with "Warning message". Function authors can also communicate with their users with `print()` or `cat()`, but I think that's a bad idea because it's hard to capture and selectively ignore this sort of output. Printed output is not a condition, so you can't use any of the useful condition handling tools you'll learn about below.

Condition handling tools, like `try()`, `tryCatch()` and `withCallingHandlers()`, allow you to take specific actions when a condition occurs. For example, if you're fitting many models, you might want to continue fitting the others even if one fails to converge. R offers an exceptionally powerful condition handling system based on ideas from Common Lisp, but it's currently not very well documented or often used. This chapter will introduce you to the most important basics, but if you want to learn more, I recommend the following two sources:

- A prototype of a condition system for R[1] by Robert Gentleman and Luke Tierney. This is describes an early version of R's condition system. The implementation has changed somewhat since this was written, but it provides a good overview of how the pieces fit together, and some motivation for the design.
- Beyond Exception Handling: Conditions and Restarts[2] by Peter Seibel. This describes exception handling in Lisp, which happens to be very similar to R's approach. It provides useful motivation and more sophisticated examples. I have provided an R translation of the chapter at http://adv-r.had.co.nz/beyond-exception-handling.html.

The chapter concludes with a discussion of "defensive" programming, avoiding common errors before they occur. You'll spend a little more time writing your code, but you'll save time in the long run by reducing errors and providing more informative error messages. The basic principle of defensive programming is to "fail fast", to raise an error as soon as you know there's something wrong, rather than trying to silently struggle through. In R, this has three particular applications: checking that inputs are correct, avoiding non-standard evaluation, and avoiding functions that can return different types of output.

[1] http://homepage.stat.uiowa.edu/~luke/R/exceptions/simpcond.html

[2] http://www.gigamonkeys.com/book/beyond-exception-handling-conditions-and-restarts.html

Debugging techniques

Finding your bug is a process of confirming the many things that you believe are true — until you find one which is not true.
— Norm Matloff

Debugging code is challenging. R provides some useful tools, which we'll discuss in the next section, but if you have a good technique, you can still productively debug a problem with just `print()`. Debugging is hard, and many bugs have a unique aspect that make them hard to find, let alone describe a general process. Indeed, if the problem was obvious, you would have avoided the bug in the first place.

The following debugging process is by no means a panacea, but will hopefully help you organise your thought process when debugging. There are four key components of the process:

1. **Realise that you have a bug**

   If you're reading this chapter, you've probably already completed this step. But this is a surprisingly important step: you can't fix a bug until you're aware of it. This is one reason why automated test suites are so important when producing high-quality code. Automated testing is unfortunately outside the scope of this book, but you can read some notes about it at http://adv-r.had.co.nz/Testing.html.

2. **Make it repeatable**

   Once you've determined you have a bug, you need to be able to recreate it on command. This can be time consuming, but if you can't consistently recreate the bug, then it's extremely difficult to isolate why it's occuring, and it's impossible to confirm that you've fixed it.

   Generally, you will start with a big block of code that you know causes the error and then slowly whittle it down to get to the smallest possible snippet that still causes the error. Binary search is particular useful for this. To do a binary search, you repeatedly remove half of the code. The bug will either re-appear or not; and either way you've reduced the amount of code to look through by half. This allows you to quickly narrow down the bug.

   If it takes a long time to generate the bug, it's also worthwhile figuring how to make it faster. The more quickly you can recreate the bug, the more quickly you'll figure out the cause.

   As you work on creating a minimal example, you'll also discover similar inputs that don't cause the bug. Make a note of them: they will be helpful when diagnosing the cause of the bug.

   If you're using automated testing, this is a good time to create an automated test case. If your existing test coverage is low, take the opportunity to add some nearby tests to ensure that existing good behaviour is preserved, thus reducing your chances of creating a new bug.

3. **Figure out where it is**

   If you're lucky, one of the tools in the following section will allow you to quickly navigate to the line of code that's causing the bug. Usually, however, you'll have to think a bit more about the problem. It's a great idea to adopt the scientific process. Generate hypotheses, design experiments to test them and then record your results. This seems like a lot of work, but a systematic approach will end up saving you time. I often end up wasting too much time trying to use my intuition to solve a bug ("oh, it must be an off-by-one error, so I'll just subtract 1 here"), when I would have been better off taking a systematic approach.

4. **Fix it and test it.**

   Once you've found the bug, you need to figure out how to fix it, and then check that the fix actually worked. Again, it's very useful to have automated tests in place, so that you can ensure that you've actually fixed the bug, and you haven't created any new bugs in the process. In the absence of automated tests, make sure to carefully record the correct output, and check against the inputs that previously failed.

Debugging tools

As well as a broad strategy to follow when debugging code, you also need some specific tools to apply. In this section you'll learn about tools provided with R and the RStudio IDE. Rstudio's integrated debugging support makes life easier, but it mostly exposes existing R tools in a user friendly way. I'll show you both the Rstudio way and the regular R way so that you can work with whatever environment you have. You may also want to refer to the official Rstudio debugging documentation[3] which will always reflect the tools in the latest version of Rstudio.

There are three key debugging tools:

- The Rstudio error inspector and `traceback()` which list the sequence of calls that lead to the error.
- Rstudio's "Rerun on debug" tool and `options(error = browser)` which enter an interactive session where the error occurred.
- Rstudio's breakpoints and `browser()` which enter an interactive session at an arbitrary code location.

[3]http://www.rstudio.com/ide/docs/debugging/overview

I'll explain each tool in more detail below.

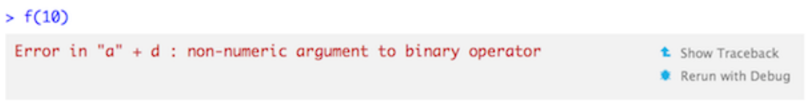
You shouldn't need to use these tools when writing new functions. If you find yourself using them frequently with new code, you may want to reconsider your approach. Instead of trying to write one big function all at once, work interactively on small pieces. If you start small, you can quickly identify why something doesn't work, rather than struggling to identify the problem in a large function.

Determining the sequence of calls

The first tool is the **call stack**, the sequence of calls that lead up to an error. Here's a simple an example: you can see that `f()` calls `g()` calls `h()` calls `i()` which adds together a number and a string creating a error:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

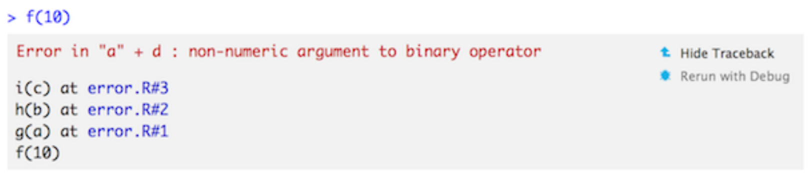
When we run this code in Rstudio we see:



```
> f(10)
Error in "a" + d : non-numeric argument to binary operator
```

Figure 8.1: Initial traceback display

If you click "Show traceback" you see:



```
> f(10)
Error in "a" + d : non-numeric argument to binary operator

i(c) at error.R#3
h(b) at error.R#2
g(a) at error.R#1
f(10)
```

Figure 8.2: Traceback display after clicking "show traceback"

If you're not using Rstudio, you can use `traceback()` to get the same information:

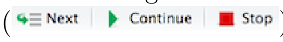
```
traceback()
# 4: i(c) at error.R#3
# 3: h(b) at error.R#2
# 2: g(a) at error.R#1
# 1: f(10)
```

You read the call stack from bottom to top: the initial call is `f()`, which eventually calls `i()` which triggers the error. If you're calling code that you `source()`d into R, the traceback will also display the location of the function, in the form `filename.r#linenumber`. These are clickable in Rstudio, and will take you to the corresponding line of code in the editor.

Sometimes this is enough information to let you track down the error and fix it. However, it's usually not: it shows you where the error occurred, but not why. The next useful tool is the interactive debugger, which allows you to pause execution of a function and interactively explore its state.

Browsing on error

The easiest way to enter the interactive debugger is through RStudio's "Rerun with debug" tool. This reruns the command that created the error, pausing execution where the error occurred. You're now in an interactive state inside the function, and you can interact with any object defined there. You'll see the corresponding code in the editor (with the statement that will be run next highlighted), objects in the current environment in the "Environment" pane, the call stack in a "Traceback" pane, and you can run arbitrary R code in the console.

As well as any regular R function, there are a few special commands you can use in debug mode. You can access them either with the Rstudio toolbar (

) or with the keyboard:

- Next, `n`: executes the next step in the function. Be careful if you have a variable named `n`; to print it you'll need to do `print(n)`.
- Continue, `c`: leaves interactive debugging and continues regular execution of the function. This is useful if you've fixed the bad state and want to check that the function proceeds correctly.
- Stop, `Q`: stops debugging, terminates the function and return to the global workspace. Use this once you've figured out where the problem is, and you're ready to fix it and reload the code.

There are two other slightly less useful commands that aren't available in the toolbar:

- Enter: repeats the previous command. I find this too easy to activate accidentally, so I turn it off using `options(browserNLdisabled = TRUE)`.
- `where`: prints stack trace of active calls (the interactive equivalent of `traceback`).

To enter this style of debugging outside of Rstudio, you can use the `error` option which specifies a function to run when an error occurs. The function most similar to Rstudio's debug is `browser()`: this will start an interactive console in the environment where the error occurred. Use `options(error = browser)` to turn it on, re-run the previous command, then use `options(error = NULL)` to return to the default error behaviour. You could automate this with the `browseOnce()` function as defined below:

```r
browseOnce <- function() {
  old <- getOption("error")
  function() {
    options(error = old)
    browser()
  }
}
options(error = browseOnce())

f <- function() stop("!")
# Enters browser
f()
# Runs normally
f()
```

(You'll learn more about functions that return functions in Functional programming[4].)

There are two other useful functions that you can use with the `error` option:

- `recover` is a step up from `browser`, as it allows you to enter the environment of any of the calls in the call stack. This is useful because often the root cause of the error is a number of calls back.

- `dump.frames` is an equivalent to `recover` for non-interactive code. It creates a `last.dump.rda` file in the current working directory. Then, in a later interactive R session, you load that file, and use `debugger()` to enter an interactive debugger with the same interface as `recover()`. This allows interactive debugging of batch code.

  ```r
  # In batch R process ----
  dump_and_quit <- function() {
    # Save debugging info to file last.dump.rda
    dump.frames(to.file = TRUE)
    # Quit R with error status
    q(status = 1)
  ```

[4]functional-programming.html

```
}
options(error = dump_and_quit)

# In a later interactive session ----
load("last.dump.rda")
debugger()
```

To reset error behaviour to the default, use `options(error = NULL)`. Then errors will print a message and abort function execution.

Browsing arbitrary code

As well as entering an interactive console on error, you can enter it at an arbitrary code location by using either an Rstudio breakpoint or `browser()`. You can set a breakpoint in Rstudio by clicking to the left of the line number, or pressing `Shift + F9`, or equivalently, add `browser()` when you want execution to pause. Breakpoints behave similarly to `browser()` but they are easier to set (one click instead of nine key presses), and you don't run the risk of accidentally including a `browser()` statement in your source code. There are two small downsides to breakpoints:

- There are few unusual situations in which breakpoints will not work: read breakpoint troubleshooting[5] for more details.
- Rstudio currently does not support conditional breakpoints, whereas you can always put `browser()` inside an `if` statement.

As well as adding `browser()` yourself, there are two functions that will add it to code for you:

- `debug()` inserts a browser statement in the first line of the specified function. `undebug()` will remove it, or you can use `debugonce()` to browse only on the next run.
- `utils::setBreakpoint()` works similarly, but instead of taking a function name, it takes a file name and line number and finds the appropriate function for you.

These two functions are both special cases of `trace()`, which inserts arbitrary code at any position in an existing function. `trace()` is occasionally useful when you're debugging code that you don't have the source for. To remove tracing from a function, use `untrace()`. You can only perform one trace per function, but that one trace can call multiple functions.

[5] http://www.rstudio.com/ide/docs/debugging/breakpoint-troubleshooting

The call stack: `traceback()`, `where` and `recover()`.

Unfortunately the call stacks printed by `traceback()`, `browser()` + `where` and `recover()` are not consistent. The following table shows how the call stacks from a simple nested set of calls are displayed by the three tools.

`traceback()`	`where`	`recover()`
4: stop("Error")	where 1: stop("Error")	1: f()
3: h(x)	where 2: h(x)	2: g(x)
2: g(x)	where 3: g(x)	3: h(x)
1: f()	where 4: f()	

Note that numbering is different between `traceback()` and `where`, and `recover()` displays calls in the opposite order, and omits the call to `stop()`. Rstudio displays calls in the same order as `traceback()` but omits the numbers.

Other types of failure

There are other ways for a function to fail apart from throwing an error or returning an incorrect result.

- A function may generate an unexpected warning. The easiest way to track down warnings is to convert them into errors with `options(warn = 2)` and use the regular debugging tools. When you do this you'll see some extra calls in the call stack, like `doWithOneRestart()`, `withOneRestart()`, `withRestarts()` and `.signalSimpleWarning()`. Ignore these: they are internal functions used to turn warnings into errors.

- A function may generate an unexpected message. There's no built in tool to help solve this problem, but it's possible to create one:

  ```
  message2error <- function(code) {
    withCallingHandlers(code, message = function(e) stop(e))
  }

  f <- function() g()
  g <- function() message("Hi!")
  g()
  # Error in message("Hi!"): Hi!
  message2error(g())
  traceback()
  ```

```
# 10: stop(e) at #2
# 9: (function (e) stop(e))(list(message = "Hi!\n", call = message("Hi!")))
# 8: signalCondition(cond)
# 7: doWithOneRestart(return(expr), restart)
# 6: withOneRestart(expr, restarts[[1L]])
# 5: withRestarts()
# 4: message("Hi!") at #1
# 3: g()
# 2: withCallingHandlers(code, message = function(e) stop(e)) at #2
# 1: message2error(g())
```

As with warnings, you'll need to ignore some of the calls on the tracback (i.e. the first two and the last seven).

- A function might never return. This is particularly hard to debug automatically, but sometimes terminating the function and looking at the call stack is informative. Otherwise, use the basic debugging strategies described above.

- The worst scenario is that your code might crash R completely, leaving you with no way to interactively debug your code. This indicates a bug in underlying C code and is hard to debug. Sometimes an interactive debugger, like `gdb`, can be useful, but describing how to use it is beyond the scope of this book.

  If the crash is caused by base R code, post a reproducible example to R-help. If it's in a package, contact the package maintainer. If it's your own C or C++ code, you'll need to use numerous `print()` statements to narrow down the location of the bug, and then you'll need to use many more print statements to figure out which data structure doesn't have the properties that you expect.

Condition handling

Unexpected errors require interactive debugging to figure out what went wrong. Some errors, however, are expected, and you want to handle them automatically. In R, expected errors crop up most frequently when you're fitting many models to different datasets, such as bootstrap replicates. Sometimes the model might fail to fit and throw an error, but you don't want to stop everything; instead you want to fit as many models as possible and then perform diagnostics after the fact.

In R, there are three tools for handling conditions (including errors) programmatically:

- `try()` gives you the ability to continue execution even when an error occurs.

- `tryCatch()` lets you specify **handler** functions that control what happens when a condition is signalled.
- `withCallingHandlers()` is a variant of `tryCatch()` that runs its handlers in a different context. It is rarely needed, but is useful to be aware of.

The following sections describe them in more detail.

Ignore errors with `try()`

`try()` allows execution to continue even after an error has occurred. For example, normally if you run a function that throws an error, it terminates immediately and doesn't return a value:

```
f1 <- function(x) {
  log(x)
  10
}
f1("x")
#> Error:
```

However, if you wrap the statement that creates the error in `try()`, the error message will be printed but execution will continue:

```
f2 <- function(x) {
  try(log(x))
  10
}
f2()
#> [1] 10
```

You can suppress the message with `try(..., silent = TRUE)`.

To pass larger blocks of code to `try()`, wrap them in `{}`:

```
try({
  a <- 1
  b <- "x"
  a + b
})
```

You can also capture the output of the `try()` function. If successful, it will be the last result evaluated in the block (just like a function); if unsuccessful it will be an (invisible) object of class "try-error":

```
success <- try(1 + 2)
failure <- try("a" + "b")
str(success)
#>  num 3
str(failure)
#> Class 'try-error'  atomic [1:1] Error in "a" + "b" :
#>
#>   ..- attr(*, "condition")=List of 2
#>   .. ..$ message: chr "        "
#>   .. ..$ call   : language "a" + "b"
#>   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

`try()` is particularly useful when you're applying a function to multiple elements in a list:

```
elements <- list(1:10, c(-1, 10), c(T, F), letters)
results <- lapply(elements, log)
#> Warning:   NaNs

#> Error:
results <- lapply(elements, function(x) try(log(x)))
#> Warning:   NaNs
```

There isn't a built-in function for testing for this class, so we'll define one. Then you can easily find the locations of errors with `sapply()` (as discussed in the Functions chapter), and extract the successes or look at the inputs that lead to failures.

```
is.error <- function(x) inherits(x, "try-error")
succeeded <- !sapply(results, is.error)

# look at successful results
str(results[succeeded])
#> List of 3
#>  $ : num [1:10] 0 0.693 1.099 1.386 1.609 ...
#>  $ : num [1:2] NaN 2.3
#>  $ : num [1:2] 0 -Inf

# look at inputs that failed
str(elements[!succeeded])
#> List of 1
#>  $ : chr [1:26] "a" "b" "c" "d" ...
```

Another useful `try()` idiom is using a default value if an expression fails. Simply assign the default value outside the try block, and then run the risky code:

```
default <- NULL
try(default <- read.csv("possibly-bad-input.csv"), silent = TRUE)
```

The function operators chapter discusses `failwith()` which makes this pattern even easier.

Handle conditions with `tryCatch()`

`tryCatch()` is a general tool for handling conditions: as well as errors you can take different actions for warnings, messages and interrupts. You've seen errors (made by `stop()`), warnings (`warning()`) and messages (`message()`) before, but interrupts are new. They can't be generated directly by the programmer, but are raised when the user attempts to terminate execution by pressing Ctrl + Break, Escape, or Ctrl + C (depending on the platform).

With `tryCatch()` you map conditions to handlers, named functions that are passed the condition as an input. If a condition is signalled, `tryCatch` will call the first handler whose name matches one of the classes of the condition. The only useful built-in names are `error`, `warning`, `message`, `interrupt` and the catch-all `condition`. A handler function can do anything, but typically it will either return a value or create a more informative error message. For example, the `show_condition()` function below sets up handlers that return the type of condition signalled:

```
show_condition <- function(code) {
  tryCatch(code,
    error = function(c) "error",
    warning = function(c) "warning",
    message = function(c) "message"
  )
}
show_condition(stop("!"))
#> [1] "error"
show_condition(warning("?!"))
#> [1] "warning"
show_condition(message("?"))
#> [1] "message"

# If no condition is captured, tryCatch returns the value of the input
show_condition(10)
#> [1] 10
```

You can use `tryCatch()` to implement `try()`. A simple implementation is shown below: the real version is more complicated to make the error message look more like what you'd see if `tryCatch()` wasn't used. Note the use of `conditionMessage()` to extract the message associated with the original error.

```
try2 <- function(code, silent = FALSE) {
  tryCatch(code, error = function(c) {
    msg <- conditionMessage(c)
    if (!silent) message("Error: ", c)
    invisible(structure(msg, class = "try-error"))
  })
}

try2(1)
#> [1] 1
try2(stop("Hi"))
#> Error: Error in doTryCatch(return(expr), name, parentenv, handler): Hi
try2(stop("Hi"), silent = TRUE)
```

As well as returning default values when a condition is signalled, handlers can be used to make more informative error messages. For example, the following function wraps around `read.csv()` to add the file name to any errors by modifying the message stored in error condition object:

```
read.csv2 <- function(file, ...) {
  tryCatch(read.csv(file, ...), error = function(c) {
    c$message <- paste0(c$message, " ( in ", file, ")")
    stop(c)
  })
}
read.csv("code/dummy.csv")
#> Error:      'row.names'
read.csv2("code/dummy.csv")
#> Error:      'row.names' ( in code/dummy.csv)
```

Catching interrupts can be useful if you want to take special action when the user tries to abort running code. But be careful, it's easy to create a loop that you can never escape! (unless you kill R)

```
# Don't let the user interrupt the code
i <- 1
while(i < 3) {
  tryCatch({
    Sys.sleep(0.5)
    message("Try to escape")
  }, interrupt = function(x) {
    message("Try again!")
    i <<- i + 1
  })
}
```

`tryCatch()` has one other argument: `finally`, which specifies a block of code (not a function) to run regardless of whether of the initial expression succeeds or fails. This can be useful for clean up (e.g. deleting files, closing connections). This is functionally equivalent to using `on.exit()` but it can wrap smaller chunks of code than an entire function.

`withCallingHandlers()`

An alternative to `tryCatch()` is `withCallingHandlers()`. There are two main differences between the functions:

- The return value of `tryCatch()` handlers is returned by `tryCatch()`, where the return value of `withCallingHandlers()` handlers is ignored:

  ```
  f <- function() stop("!")
  tryCatch(f(), error = function(e) 1)
  #> [1] 1
  withCallingHandlers(f(), error = function(e) 1)
  #> Error: !
  ```

- The handlers in `withCallingHandlers()` are called in the context of the call that generated the condition; the handlers in `tryCatch()` are called in the context of `tryCatch()`. (`sys.calls()` is the run-time equivalent of `traceback()`, listing all calls leading to the current function.)

  ```
  f <- function() g()
  g <- function() h()
  h <- function() stop("!")

  tryCatch(f(), error = function(e) print(sys.calls()))
  # [[1]] tryCatch(f(), error = function(e) print(sys.calls()))
  # [[2]] tryCatchList(expr, classes, parentenv, handlers)
  # [[3]] tryCatchOne(expr, names, parentenv, handlers[[1L]])
  # [[4]] value[[3L]](cond)

  withCallingHandlers(f(), error = function(e) print(sys.calls()))
  # [[1]] withCallingHandlers(f(), error = function(e) print(sys.calls()))
  # [[2]] f()
  # [[3]] g()
  # [[4]] h()
  # [[5]] stop("!")
  # [[6]] .handleSimpleError(function (e) print(sys.calls()), "!", quote(h()))
  # [[7]] h(simpleError(msg, call))
  ```

  This also affects the order in which `on.exit()` is called.

These subtle differences are rarely useful, except when you're trying to capture exactly what went wrong and pass it on to another function. For most purposes, you should never need to use `withCallingHandlers()`.

Custom signal classes

One of the challenges of error handling in R is that most functions just call `stop()` with a string. That means if you want to figure out if a particular error occurred, you have to look at the text of the error message. This is error prone, not only because the text of the error might change over time, but also because many error messages are translated, so the message might be completely different to what you expect.

R has a little known and little used feature to solve this problem. Conditions are S3 classes, so you can define your own classes if you want to distinguish different types of error. Each condition signalling function, `stop()`, `warning()` and `message()` can be given either a list of strings, or a custom S3 condition object. Custom condition objects are not used very often, but are very useful because they make it possible for the user to respond to different errors in different ways. For example, "expected" errors (like a model failing to converge for some input datasets) can be silently ignored, while unexpected errors (like no disk space available) can be propagated to the user.

R doesn't come with a built-in constructor function for conditions, but we can easily add one. Conditions must contain `message` and `call` components, and may contain other useful components. When creating a new condition, it should always inherit from `condition` and one of `error`, `warning` and `message`.

```
condition <- function(subclass, message, call = sys.call(-1), ...) {
  structure(
    class = c(subclass, "condition"),
    list(message = message, call = call),
    ...
  )
}
is.condition <- function(x) inherits(x, "condition")
```

You can signal an arbitrary condition with `signalCondition()`, but nothing will happen unless you've instantiated a custom signal handler (with `tryCatch()` or `withCallingHandlers()`. Instead, use `stop()`, `warning()` or `message()` as appropriate to trigger the usual handling. R won't complain if the class of your condition doesn't match the function, but you should avoid this in real code.

```
c <- condition(c("my_error", "error"), message = "This is an error")
signalCondition(c)
```

```
# NULL
stop(c)
# Error: This is an error
warning(c)
# Warning message: This is an error
message(c)
# This is an error
```

You can then use `tryCatch()` to take different actions for different types of errors. In this example we make a convenient `custom_stop()` function that allows us to signal error conditions with arbitrary classes. In a real application, it would be better to have individual S3 constructor functions that you could document, describing the error classes in more detail.

```
custom_stop <- function(subclass, message, call = sys.call(-1), ...) {
  c <- condition(c(subclass, "error"), message, call = call, ...)
  stop(c)
}

my_log <- function(x) {
  if (!is.numeric(x))
    custom_stop("invalid_class", "my_log() needs numeric input")
  if (any(x < 0))
    custom_stop("invalid_value", "my_log() needs positive inputs")

  log(x)
}
tryCatch(
  my_log("a"),
  invalid_class = function(c) "class",
  invalid_value = function(c) "value"
)
#> [1] "class"
```

Note that when using `tryCatch()` with multiple handlers and custom classes, the first handler to match any class in the signal's class hierarchy is called, not the best match. For this reason, you need to make sure to put the most specific handlers first:

```
tryCatch(customStop("my_error", "!"),
  error = function(c) "error",
  my_error = function(c) "my_error"
)
#> [1] "error"
tryCatch(custom_stop("my_error", "!"),
  my_error = function(c) "my_error",
  error = function(c) "error"
)
#> [1] "my_error"
```

Exercises

- Compare the following two implementations of `message2error()`. What is the main advantage of `withCallingHandlers()` in this scenario? (Hint: look carefully at the traceback.)

  ```
  message2error <- function(code) {
    withCallingHandlers(code, message = function(e) stop(e))
  }
  message2error <- function(code) {
    tryCatch(code, message = function(e) stop(e))
  }
  ```

Defensive programming

Defensive programming is the art of making code fail in a well-defined manner even when something unexpected occurs. A key principle of defensive programming is to “fail fast”: as soon as you discover something is wrong, signal an error. This is more work for you as the function author, but will make it easier for the user to debug because they get errors early on, not after unexpected input has passed through several functions.

The principle of “fail fast” has three main applications in R:

- Be strict about what you accept. For example, if your function is not vectorised in its inputs, but uses functions that are, make sure to check that the inputs are scalars. You can use `stopifnot()`, the assertthat[6] package or simple `if` statements and `stop()`.

- Avoid functions that use non-standard evaluation, like `subset`, `transform`, and `with`. These functions save time when used interactively, but because they make assumptions to reduce typing, when they fail, they often fail with uninformative error messages. You can learn more about non-standard evaluation in the metaprogramming[7] chapter.

- Avoid functions that return different types of output depending on their input. The two biggest offenders are `[` and `sapply()`. Whenever subsetting a data frame in a function, you should always use `drop = FALSE`,

[6] https://github.com/hadley/assertthat
[7] Computing-on-the-language.html

otherwise you will accidentally convert 1-column data frames into vectors. Similarly, never use `sapply()` inside a function: always use the stricter `vapply()` which will throw an error if the inputs are incorrect types and return the correct type of output even for zero-length inputs.

There is a tension between interactive analysis and programming. When you're doing an analysis, you want R to do what you mean, and if it guesses wrong, you'll discover it right away and you can fix it. When you're programming, you want functions with no magic that signal errors is anything is slightly wrong or underspecified. Keep this tension in mind when writing functions: If you're making a function to facilitate interactive data analysis, feel free to guess what the analyst wants and recover from minor misspecifications automatically; if you're making a function to program with, be strict, and never make guesses about what the caller wants.

Exercises

- The goal of the `col_means()` function defined below is to compute the means of all numeric columns in a data frame.

  ```
  col_means <- function(df) {
    numeric <- sapply(df, is.numeric)
    numeric_cols <- df[, numeric]

    data.frame(lapply(numeric_cols, mean))
  }
  ```

  However, the function is not robust to unusual inputs. Look at the following results, decide which ones are incorrect, and modify `col_means` to be more robust. (Hint: there are two function calls in `col_means` that are particularly prone to problems.)

  ```
  col_means(mtcars)
  col_means(mtcars[, 0])
  col_means(mtcars[0, ])
  col_means(mtcars[, "mpg", drop = F])
  col_means(1:10)
  col_means(as.matrix(mtcars))
  col_means(as.list(mtcars))

  mtcars2 <- mtcars
  mtcars2[-1] <- lapply(mtcars2[-1], as.character)
  col_means(mtcars2)
  ```

- The following function "lags" a vector, returning a version of `x` that is `n` values behind the original. Improve the function so that (1) it returns a useful error message if `n` is not a vector, (2) it has reasonable behaviour when `n` is 0 or longer than `x`.

```
lag <- function(x, n = 1L) {
  xlen <- length(x)
  c(rep(NA, n), x[seq_len(xlen - n)])
}
```

Part II

Functional programming

Chapter 9

Functional programming

At its heart, R is a functional programming (FP) language; it focusses on the creation and manipulation of functions. R has what's known as first class functions, functions that can be:

- created without a name,
- assigned to variables and stored in lists,
- returned from functions, and
- passed as arguments to other functions.

This means that you can do anything with functions that you can do with vectors: you can create them inside other functions, pass them as arguments to functions, return them as results from functions and store multiple functions in a list. This chapter will explore the consequences of R's functional nature and introduce a new set of techniques for removing redundancy and duplication in your code. We'll start with a motivating example, showing how you can use functional programming techniques to reduce duplication in some typical code for cleaning data and summarising data. This example will introduce some of the key building blocks of functional programming, which we will then dive into in more detail:

- **Anonymous functions**, functions that don't have a name
- **Closures**, functions written by other functions
- **Lists of functions**, storing functions in a list

The chapter concludes with a case study exploring **numerical integration** showing how we can build a family of composite integration tools starting from very simple primitives. This will be a recurring theme: if we start with small building blocks that we can easily understand, when we combine them into more complex structures, we can still feel confident that they are correct.

The exposition of functional programming continues in the following two chapters: [[functionals]], which explore functions that take functions as arguments and give vectors as output, and [[function operators]], functions that both input and output functions.

Other languages

FP techniques are the core technique in FP languages, like Haskell, OCaml and F#. They are also well supported in multi-paradigm systems like Lisp, Scheme, Clojure and Scala. You can use FP techniques in modern scripting languages, like Python, Ruby and Javascript, but they tend not to be the dominant technique employed by most programmers. Java and C# provide few functional tools, and while it's possible to do FP in those languages, it tends to be a somewhat awkward fit. Similarly, for functional programming in C. Googling for "functional programming in X" will find you a tutorial in any language, but it may be syntactically awkward or used so rarely that other programmers will not understand your code.

Recently FP has experienced a surge in interest because it provides a complementary set of techniques to object oriented programming, which has been the dominant style for the last several decades. Since FP functions tend to not modify their inputs, it makes 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.

Motivation

Imagine you've loaded a data file that uses -99 to represent missing values, like the following sample dataset.

```r
# Generate a sample dataset
set.seed(1014)
df <- data.frame(replicate(6, sample(c(1:10, -99), 10, rep = T)))
names(df) <- letters[1:6]
head(df)
```

When you first start writing R code, you might write code like the following, dealing with the duplicated processing of each column with copy-and-paste:

```
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -98] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$g == -99] <- NA
```

One problem with copy-and-paste is that it's easy to make mistakes: can you spot two in the block above? The problem is that one idea, that missing values are represent as -99, is repeated many times. Repetition is bad because it allows for inconsistencies (aka bugs), and it makes the code harder to change. For example, if the representation of missing values changes from -99 to 9999, then we need to make the change in many places, not just one.

The "do not repeat yourself", or DRY, principle, was popularised by the pragmatic programmers[1], Dave Thomas and Andy Hunt. This principle states that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system". Adhering to this principle prevents bugs caused by inconsistencies, and makes software that is easier to adapt to changing requirements. The ideas of FP are important because they give us new tools to reduce duplication.

We can start applying some of the ideas of FP to our example by writing a function that fixes the missing values in a single vector:

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$e)
```

This reduces the scope for errors, but doesn't eliminate them. We've still made an error, because we've repeatedly applied our function to each column. To prevent that error from occuring we need to remove the copy-and-paste application of our function to each column. To do this, we need to combine, or compose, our function for correcting missing values with a function that does something to each column in a data frame, like `lapply()`.

`lapply()` takes three inputs: `x`, a list; `f`, a function; and '`...`, other arguments to pass to `f`. It applies the function to each element of the list and returns a

[1] http://pragprog.com/about

new list. Since data frames are also lists, `lapply()` also works on data frames. `lapply(x, f, ...)` is equivalent to the following for loop:

```
out <- vector("list", length(x))
for (i in seq_along(x)) {
  out[[i]] <- f(x[[i]], ...)
}
```

The real `lapply()` is rather more complicated since it's implemented in C for efficiency, but the essence of the algorithm is the same. `lapply()` is called a **functional**, because it takes a function as an argument. Functionals are an important part of functional programming and we'll learn more about them in the [[functionals]] chapter.

We can use `lapply()` with one small trick: rather than simply assigning the results to `df` we assign them to `df[]`, so R's usual subsetting rules take over and we get a data frame instead of a list. (If this comes as a surprise, you might want to read over the [[subsetting]] appendix)

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df[] <- lapply(df, fix_missing)
```

As well as being more compact, there are four main advantages of this code over our previous code:

- If the representation of missing values changes, we only need to change it in one place.
- There is no way for some columns to be treated differently than others.
- Our code works regardless of the number of columns in the data frame, and there is no way to miss a column because of a copy and paste error.
- It is easy to generalise this technique to a subset of our columns:

  ```
  df[1:5] <- lapply(df[1:5], fix_missing)
  ```

The key idea here is composition. We take two simple functions, one which does something to each column, and one which fixes missing values, and combine them together to fix missing values in every column. Writing simple functions than can be understood in isolation and then composed together to solve complex problems is an important technique for effective FP.

What if different columns used different indicators for missing values? You again might be tempted to copy-and-paste:

```
fix_missing_99 <- function(x) {
  x[x == -99] <- NA
  x
}
fix_missing_999 <- function(x) {
  x[x == -999] <- NA
  x
}
fix_missing_9999 <- function(x) {
  x[x == -999] <- NA
  x
}
```

But as before, it's easy to create bugs. The next functional programming tool we'll talk about helps deal with this sort of duplication: when we have multiple functions that all follow the same basic template. Closures, functions that return functions, allow us to make many functions from a template:

```
missing_fixer <- function(na_value) {
  function(x) {
    x[x == na_value] <- NA
    x
  }
}
fix_missing_99 <- missing_fixer(-99)
fix_missing_999 <- missing_fixer(-999)
fix_missing_9999 <- missing_fixer(-9999)
```

(In this case, you could argue that we should just add another argument:

```
fix_missing <- function(x, na.value) {
  x[x == na.value] <- NA
  x
}
```

That's a reasonable solution here, but it doesn't work so well in every situation. We'll see more compelling uses for closures later in the chapter.)

Let's now consider a new problem: once we've cleaned up our data, we might want to compute the same set of numerical summaries for each variable. We could write code like this:

```
mean(df$a)
median(df$a)
sd(df$a)
```

```
mad(df$a)
IQR(df$a)

mean(df$b)
median(df$b)
sd(df$b)
mad(df$b)
IQR(df$b)
```

But we'd be better off identifying the sources of duplication and then removing them. Take a minute or two to think about how you might tackle this problem before reading on.

One approach would be to write a summary function and then apply it to each column:

```
summary <- function(x) {
  c(mean(x), median(x), sd(x), mad(x), IQR(x))
}
lapply(df, summary)
```

But there's still some duplication here. If we make the summary function slightly more realistic, it's easier to see the duplication:

```
summary <- function(x) {
 c(mean(x, na.rm = TRUE),
   median(x, na.rm = TRUE),
   sd(x, na.rm = TRUE),
   mad(x, na.rm = TRUE),
   IQR(x, na.rm = TRUE))
}
```

All five functions are called with the same arguments (`x` and `na.rm`) which we had to repeat five times. As before, this duplication makes our code fragile: it makes it easier to introduce bugs and harder to adapt to changing requirements.

We can take advantage of another functional programming technique, storing functions in lists, to remove this duplication:

```
summary <- function(x) {
  funs <- c(mean, median, sd, mad, IQR)
  lapply(funs, function(f) f(x, na.rm = TRUE))
}
```

The remainder of this chapter will discuss these techniques in more detail. But before we can start on those more complicated techniques, we need to start by revisiting a simple functional programming tool, anonymous functions.

Anonymous functions

In R, functions are objects in their own right. They aren't automatically bound to a name and R doesn't have a special syntax for creating named functions, unlike C, C++, Python or Ruby. You might have noticed this already, because when you create a function, you use the usual assignment operator to give it a name.

Given the name of a function, like `"mean"`, it's possible to find the function using `match.fun()`. You can't do the opposite: given the object `f <- mean`, there's no way to find its name. Not all functions have a name, and some functions have more than one name. Functions that don't have a name are called **anonymous functions**.

We use anonymous functions when it's not worth the effort of creating a named function:

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

Unfortunately the default R syntax for anonymous functions is quite verbose. To make things a little more concise, the `pryr` packages provides `f()`:

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

I'm not still sure whether I like this style or not, but it sure is compact! Other similar ideas are implemented in `gsubfn::fn()` and `ptools::fun()`.

Like all functions in R, anoynmous functions have `formals()`, a `body()`, and a parent `environment()`:

```
formals(function(x = 4) g(x) + h(x))
body(function(x = 4) g(x) + h(x))
environment(function(x = 4) g(x) + h(x))
```

You can call anonymous functions directly, but the code is a little tricky to read because you must use parentheses in two different ways: to call a function, and to make it clear that we want to call the anonymous function `function(x) 3`, not inside our anonymous function call a function called `3` (which isn't a valid function name!):

```
(function(x) x + 3)(10)

# Exactly the same as
f <- function(x) x + 3
f(10)

# Doesn't do what you expect
function(x) 3()
```

You can supply arguments to anonymous functions in all the usual ways (by position, exact name and partial name) but if you find yourself doing this, it's probably a sign that your function needs a name.

One of the most common uses for anonymous functions is to create closures, functions made by other functions. Closures are described in the next section.

Exercises

- Use `lapply()` and an anonymous function to find the coefficient of variation (the standard deviation divided by the mean) for all columns in the `mtcars` dataset
- Use `integrate()` and an anonymous function to find the area under the curve of:
- `y = x ^ 2 - x`, x in [0, 10]
- `y = sin(x) + cos(x)`, x in [-pi, pi]
- `y = exp(x) / x`, x in [10, 20]

Use wolframalpha[2] to check your answers.

- 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?

Introduction to closures

"An object is data with functions. A closure is a function with data." — John D Cook[3]

[2]http://www.wolframalpha.com/
[3]http://twitter.com/JohnDCook/status/29670670701

One use of anonymous functions is to create small functions that it's not worth naming; the other main use of anonymous functions is to create closures, functions written by functions. Closures are so called because they **enclose** the environment of the parent function, and can access all variables in the parent. This is useful because it allows us to have two levels of parameters. One level of parameters (the parent) controls how the function works; the other level (the child) does the work. The following example shows how we can use this idea to generate a family of power functions. The parent function (`power()`) creates child functions (`square()` and `cube()`) that do the work.

```
power <- function(exponent) {
  function(x) x ^ exponent
}

square <- power(2)
square(2)
square(4)

cube <- power(3)
cube(2)
cube(4)
```

In R, almost every function is a closure, because all functions remember the environment in which they are created, typically either the global environment, if it's a function that you've written, or a package environment, if it's a function that someone else has written. The only exception are primitive functions, which call to C directly.

When you print a closure, you don't see anything terribly useful:

```
square
cube
```

That's because the function itself doesn't change; it's the enclosing environment, `environment(square)`, that's different. One way to see the contents of the environment is to convert it to a list:

```
as.list(environment(square))
as.list(environment(cube))
```

Another way to see what's going on is to use `pryr::unenclose()`, which substitutes the variables defined in the enclosing environment into the original functon:

```
library(pryr)
unenclose(square)
unenclose(cube)
```

Note that the parent environment of the closure is the environment created when the parent function is called:

```
power <- function(exponent) {
  print(environment())
  function(x) x ^ exponent
}
zero <- power(0)
environment(zero)
```

This environment normally disappears once the function finishes executing, but because we return a function, the environment is captured and attached to the new function. Each time we re-run `power()` a new environment is created, so each function produced by power is independent.

Closures are useful for making function factories, and are one way to manage mutable state in R.

Function factories

We've already seen two example of function factories, `missing_fixer()` and `power()`. In both these cases using a function factory instead of a single function with multiple arguments has little, if any, benefit. Function factories are most useful when:

- the different levels are more complex, with multiple arguments and complicated bodies
- some work only needs to be done once, when the function is generated

INSERT USEFUL EXAMPLE HERE

We'll see another compelling use of function factories when we learn more about [[functionals]]; they are particularly well suited to maximum likelihood problems.

Mutable state

Having variables at two levels makes it possible to maintain state across function invocations, because while the function environment is refreshed every time, its parent environment stays constant. The key to managing variables at different levels is the double arrow assignment operator (`<<-`). Unlike the usual single arrow assignment (`<-`) that always assigns in the current environment, the double arrow operator will keep looking up the chain of parent environments until it finds a matching name. ([Environments] has more details on how it works)

Together, a static parent environment and `<<-` make it possible to maintain state across function calls. The following example shows a counter that records how many times a function has been called. Each time `new_counter` is run, it creates an environment, initialises the counter `i` in this environment, and then creates a new function.

```
new_counter <- function() {
  i <- 0
  function() {
    i <<- i + 1
    i
  }
}
```

The new function is a closure, and its enclosing environment is the usually temporary environment created when `new_counter` is run. When the closures `counter_one` and `counter_two` are run, each one modifies the counter in a different enclosing environment and so maintain different counts.

```
counter_one <- new_counter()
counter_two <- new_counter()

counter_one() # -> [1] 1
counter_one() # -> [1] 2
counter_two() # -> [1] 1
```

We can use our environment inspection tools to see what's going on here:

```
as.list(environment(counter_one))
as.list(environment(counter_two))
```

The counters get around the "fresh start" limitation by not modifying variables in their local environment. Since the changes are made in the unchanging parent (or enclosing) environment, they are preserved across function calls.

What happens if we don't use a closure? What happens if we only use `<-` instead of `<<-`? Make predictions about what will happen if you replace `new_counter()` with each variant below, then run the code and check your predictions.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

Modifying values in a parent environment is an important technique because it is one way to generate "mutable state" in R. Mutable state is normally hard to achieve, because every time it looks like you're modifying an object, you're actually creating a copy and modifying that. That said, if you do need mutable objects, except in the simplest of cases, it's usually better to use the RC OO system. RC objects are easier to document, and provide easier ways to inherit behaviour.

The power of closures is tightly coupled to [[functionals]] and [[function operators]], and you'll see many more examples of closures in those two chapters. The following section disucsses the remaining important property of functions: the ability to store them in a list.

Exercises

- What does the following statistical function do? What would be a better name for it? (The existing name is a bit of a hint)

  ```
  bc <- function(lambda) {
    if (lambda == 0) {
      function(x) log(x)
    } else {
      function(x) (x ^ lambda - 1) / lambda
    }
  }
  ```

- Create a function that creates functions that compute the ith central moment[4] of a numeric vector. You can test it by running the following code:

  ```
  m1 <- moment(1)
  m2 <- moment(2)
  ```

[4] http://en.wikipedia.org/wiki/Central_moment

```
x <- runif(100)
stopifnot(all.equal(m1(x), mean(x)))
stopifnot(all.equal(m2(x), var(x) * 99 / 100))
```

- What does `approxfun()` do? What does it return?
- What does the `ecdf()` function do? What does it return?
- Create a function `pick()`, that takes an index, `i`, as an argument and returns a function an argument `x` that subsets `x` with `i`.

```
lapply(mtcars, pick(5))
# should do the same this as
lapply(mtcars, function(x) x[[5]])
```

Lists of functions

In R, functions can be stored in lists. Instead of giving a set of functions related names, you can store them in a list. This makes it easier to work with groups of related functions, in the same way a data frame makes it easier to work with groups of related vectors.

We'll start with a simple example: benchmarking, when you are comparing the performance of multiple approaches to the same problem. For example, if you wanted to compare a few approaches to computing the mean, you could store each approach (function) in a list:

```
compute_mean <- list(
  base = function(x) mean(x),
  sum = function(x) sum(x) / length(x),
  manual = function(x) {
    total <- 0
    n <- length(x)
    for (i in seq_along(x)) {
      total <- total + x[i] / n
    }
    total
  }
)
```

Calling a function from a list is straightforward: just get it out of the list first:

```
x <- runif(1e5)
system.time(compute_mean$base(x))
system.time(compute_mean[[2]](x))
system.time(compute_mean[["manual"]](x))
```

If we want to call each functions to check that we've implemented them correctly and they return the same answer, we can use `lapply()`, either with an anonymous function, or an equivalent named function.

```
lapply(compute_mean, function(f, ...) f(...), x)

call_fun <- function(f, ...) f(...)
lapply(compute_mean, call_fun, x)
```

If we want to time how long each function takes, we can combine lapply with `system.time()`:

```
lapply(compute_mean, function(f) system.time(f(x)))
```

Coming back to our original motivating example, another use case of lists of functions is summarising an object in multiple ways. We could store each summary function in a list, and then run them all with `lapply()`:

```
funs <- list(
  sum = sum,
  mean = mean,
  median = median
)
lapply(funs, function(f) f(1:10))
```

What if we wanted our summary functions to automatically remove missing values? One approach would be make a list of anonymous functions that call our summary functions with the appropriate arguments:

```
funs2 <- list(
  sum = function(x, ...) sum(x, ..., na.rm = TRUE),
  mean = function(x, ...) mean(x, ..., na.rm = TRUE),
  median = function(x, ...) median(x, ..., na.rm = TRUE)
)
```

But this leads to a lot of duplication - each function is almost identical apart from a different function name. We could write a closure to abstract this away:

Instead, we could modify our original `lapply()` call:

```
lapply(funs, function(f) f(x))
lapply(funs, function(f) f(x, na.rm = TRUE))

# Or use a named function instead of an anonymous function
remove_missings <- function(f) {
  function(...) f(..., na.rm = TRUE)
}
funs2 <- lapply(funs, remove_missings)
```

Moving lists of functions to the global environment

From time to time you may want to create a list of functions that you want to be available to your users without having to use a special syntax. For a simple example, imagine you want to make it easy to create HTML with code, by mapping each HTML tag to an R function. The following simple example creates functions for `<p>` (paragraphics), `<b>` (bold), `<i>` (italics), and `<img>` (images). Note the use of a closure function factory to produce the text for `<p>`, `<b>` and `<i>` tags.

```
simple_tag <- function(tag) {
  function(...) paste0("<", tag, ">", paste0(...), "</", tag, ">")
}
html <- list(
  p = simple_tag("p"),
  b = simple_tag("b"),
  i = simple_tag("i"),
  img = function(path, width, height) {
    paste0("<img src='", path, "' width='", width, "' height = '", height, '" />')
  }
)
```

We store the functions in a list because we don't want them to be available all the time: the risk of a conflict between an existing R function and an HTML tag is high. However, keeping them in a list means that our code is more verbose than necessary:

```
html$p("This is ", html$b("bold"), ", ", html$i("italic"), " and ",
   html$b(html$i("bold italic")), " text")
```

We have three options to eliminate the use of `html$`, depending on how long we want the effect to last:

- For a very temporary effect, we can use a `with()` block:

  ```
  with(html, p("This is ", b("bold"), ", ", i("italic"), " and ",
    b(i("bold italic")), " text"))
  ```

- For a longer effect, we can use `attach()` to add the functions in `html` in to the search path. It's possible to undo this action using `detach`:

  ```
  attach(html)
  p("This is ", b("bold"), ", ", i("italic"), " and ",
    b(i("bold italic")), " text")
  detach(html)
  ```

- Finally, we could copy the functions into the global environment with `list2env()`. We can undo this action by deleting the functions after we're done.

  ```
  list2env(html, environment())
  p("This is ", b("bold"), ", ", i("italic"), " and ",
    b(i("bold italic")), " text")
  rm(list = names(html), envir = environment())
  ```

I recommend the first option because it makes it very clear what's going on, and when code is being executed in a special context.

Exercises

- Implement a summary function that works like `base::summary()`, but takes a list of functions to use to compute the summary. Modify the function so it returns a closure, making it possible to use it as a function factory.
- Create a named list of all base functions. Use `ls()`, `get()` and `is.function()`. Use that list of functions to answer the following questions:
  - Which base function has the most arguments?
  - How many base functions have no arguments?
- Which of the following commands is `with(x, f(z))` equivalent to?
  (a) `x$f(x$z)`
  (b) `f(x$z)`
  (c) `x$f(z)`
  (d) `f(z)`

Case study: numerical integration

To conclude this chapter, we will develop a simple numerical integration tool, and along the way, illustrate the use of many properties of first-class functions. Each step is driven by a desire to make our approach more general and to reduce duplication. The idea behind numerical integration is simple: we want to find the area under the curve by approximating a complex curve with simpler components.

The two simpliest approaches are the **midpoint** and **trapezoid** rules; the mid point rule approximates a curve by a rectangle, and the trapezoid rule by a trapezoid. Each takes a function we want to integrate, `f`, and a range to integrate over, from `a` to `b`. For this example we'll try to integrate `sin x` from 0 to pi, because it has a simple answer: 2.

```r
midpoint <- function(f, a, b) {
  (b - a) * f((a + b) / 2)
}

trapezoid <- function(f, a, b) {
  (b - a) / 2 * (f(a) + f(b))
}

midpoint(sin, 0, pi)
trapezoid(sin, 0, pi)
```

Neither of these functions gives a very good approximation, so we'll do what we normally do in calculus: break up the range into smaller pieces and integrate each piece using one of the simple rules. This is called **composite integration**, and we'll implement it with two new functions:

```r
midpoint_composite <- function(f, a, b, n = 10) {
  points <- seq(a, b, length = n + 1)
  h <- (b - a) / n

  area <- 0
  for (i in seq_len(n)) {
    area <- area + h * f((points[i] + points[i + 1]) / 2)
  }
  area
}

trapezoid_composite <- function(f, a, b, n = 10) {
  points <- seq(a, b, length = n + 1)
  h <- (b - a) / n

  area <- 0
  for (i in seq_len(n)) {
    area <- area + h / 2 * (f(points[i]) + f(points[i + 1]))
  }
  area
}

midpoint_composite(sin, 0, pi, n = 10)
midpoint_composite(sin, 0, pi, n = 100)
trapezoid_composite(sin, 0, pi, n = 10)
trapezoid_composite(sin, 0, pi, n = 100)

mid <- sapply(1:20, function(n) midpoint_composite(sin, 0, pi, n))
trap <- sapply(1:20, function(n) trapezoid_composite(sin, 0, pi, n))
matplot(cbind(mid = mid, trap))
```

But notice that there's a lot of duplication across `midpoint_composite` and `trapezoid_composite`: they are basically the same apart from the internal rule used to integrate over a simple range. Let's extract out a general composite integrate function:

```
composite <- function(f, a, b, n = 10, rule) {
  points <- seq(a, b, length = n + 1)

  area <- 0
  for (i in seq_len(n)) {
    area <- area + rule(f, points[i], points[i + 1])
  }

  area
}

midpoint_composite(sin, 0, pi, n = 10)
composite(sin, 0, pi, n = 10, rule = midpoint)
composite(sin, 0, pi, n = 10, rule = trapezoid)
```

This function now takes two functions as arguments: the function to integrate, and the integration rule to use for simple ranges. We can now add even better rules for integrating small ranges:

```
simpson <- function(f, a, b) {
  (b - a) / 6 * (f(a) + 4 * f((a + b) / 2) + f(b))
}

boole <- function(f, a, b) {
  pos <- function(i) a + i * (b - a) / 4
  fi <- function(i) f(pos(i))

  (b - a) / 90 *
    (7 * fi(0) + 32 * fi(1) + 12 * fi(2) + 32 * fi(3) + 7 * fi(4))
}
```

Let's compare these different approaches.

```
expt1 <- expand.grid(
  n = 5:50,
  rule = c("midpoint", "trapezoid", "simpson", "boole"),
  stringsAsFactors = F)

abs_sin <- function(x) abs(sin(x))
run_expt <- function(n, rule) {
  composite(abs_sin, 0, 4 * pi, n = n, rule = match.fun(rule))
}

library(plyr)
res1 <- mdply(expt1, run_expt)

library(ggplot2)
qplot(n, V1, data = res1, colour = rule, geom = "line")
```

It turns out that the midpoint, trapezoid, Simpson and Boole rules are all examples of a more general family called Newton-Cotes rules. (They are polynomials of increasing complexity). We can take our integration one step further by extracting out this commonality to produce a function that can generate any general Newton-Cotes rule:

```
# http://en.wikipedia.org/wiki/Newton%E2%80%93Cotes_formulas
newton_cotes <- function(coef, open = FALSE) {
  n <- length(coef) + open

  function(f, a, b) {
    pos <- function(i) a + i * (b - a) / n
    points <- pos(seq.int(0, length(coef) - 1))

    (b - a) / sum(coef) * sum(f(points) * coef)
  }
}

trapezoid <- newton_cotes(c(1, 1))
midpoint <- newton_cotes(1, open = TRUE)
simpson <- newton_cotes(c(1, 4, 1))
boole <- newton_cotes(c(7, 32, 12, 32, 7))
milne <- newton_cotes(c(2, -1, 2), open = TRUE)

expt1 <- expand.grid(n = 5:50, rule = names(rules),
  stringsAsFactors = FALSE)
run_expt <- function(n, rule) {
  composite(abs_sin, 0, 4 * pi, n = n, rule = rules[[rule]])
}
```

Mathematically, the next step in improving numerical integration is to move from a grid of evenly spaced points to a grid where the points are closer together near the end of the range, such as **Gaussian quadrature**. That's beyond the scope of this case study, but you would use similar techniques to add it.

Exercises

- Instead of creating individual fuctions `midpoint()`, `trapezoid()`, `simpson()` etc, we could store them in a list. If we do that, how does the code change? Can you create the list of functions from a list of coefficients for the Newton-Cotes formulae?
- The tradeoff in integration rules is that more complex rules are slower to compute, but need fewer pieces. For `sin()` in the range [0, pi], determine the number of pieces needed to for each rule to be equally accurate. Illustrate your results with a graph. How do they change for different functions? `sin(1 / x^2)` is particularly challenging.
- For each of the Newton-Cotes rules, how many pieces do you need to get within 0.1% of the true answer for `sin()` in the range [0, pi]. Write a function that determines that automatically for any function (hint: look at `optim()` and construct a one-argument function with closures)

Chapter 10

Functionals

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[1]

Higher-order functions encompass any functions that either take a function as an input or return a function as output. We've already seen closures, functions returned by another function. The complement to a closure is a **functional**, a function that takes a function as an input and returns a vector as output.

Here's a simple functional, it takes an input function and calls it with some random input:

```
randomise <- function(f) f(runif(1e3))
randomise(mean)
#> [1] 0.5029
randomise(sum)
#> [1] 493.6
```

This function is not terribly useful, but it illustrates the basic idea: since functions are first class objects in R, there's no difference between calling a function with a vector or function as input. The chances are that you've already used a functional: the most frequently used are `lapply()`, `apply()` and `tapply()`. These three functions all take a function as input (among other things) and give a vector as output.

[1] http://www.stroustrup.com/Software-for-infrastructure.pdf

Many functionals (like `lapply()`) offer alternatives to for loops. For loops have a bad rap in R, and some programmers try to eliminate them at all costs. The performance story is a little more complicated than what you might have heard (we'll explore that in the [[performance]] chapter); the real downside of for loops is that they're not very expressive. A for loop conveys that you're iterating over something, but it doesn't communicate the higher-level task you're trying to complete. Functionals are not as general as for loops, but by being more specific they allow you to communicate more clearly. A functional allows you to say I want to transform each element of this list, or each row of this array.

As well as more clearly communicating intent, functionals reduce the chances of bugs, and can be more efficient. Both of these features occur because functionals are used by many people, so they will be well tested, and may have been implemented with an eye to performance. For example, many functionals in base R are written in C, and often use a few tricks to get extra performance.

As well as replacements for for loops, functionals do play other roles. They are also useful tools for encapsulating common data manipulation tasks, the split-apply-combine pattern; for thinking "functionally"; and for working with mathematical functions. In this chapter, you'll learn about:

- Functionals that replace a common pattern of for-loop use, like `lapply`, `vapply` and `Map`.
- Functionals for manipulating common R data structures, like `apply`, `split`, `tapply` and the plyr package.
- Popular functionals from other programming languages, like `Map`, `Reduce` and `Filter`.
- Mathematical functionals, like `integrate`, `uniroot`, and `optim`.

We'll also talk about how (and why) you might convert loop to use a functional. The chapter concludes with a case study where we take simple scalar addition and use functionals to build a complete family of addition functions including vectorised addition, sum, cumulative sum, and row- and column-wise summation.

The focus in this chapter is on clear communication with your code, and developing tools to solve wide classes of problems. This will not always produce the fastest code, but it is a mistake to focus on speed until you know it will be a problem. Once you do have clear, correct code you can make it fast using the techniques in the [[performance]] chapter.

My first functional: `lapply()`

The simplest functional is `lapply()`, which you may already be familiar with. `lapply()` takes a function and applies it to each element of a list, saving the results back into a list. `lapply()` is the building block for many other functionals, so it's important to understand how it works

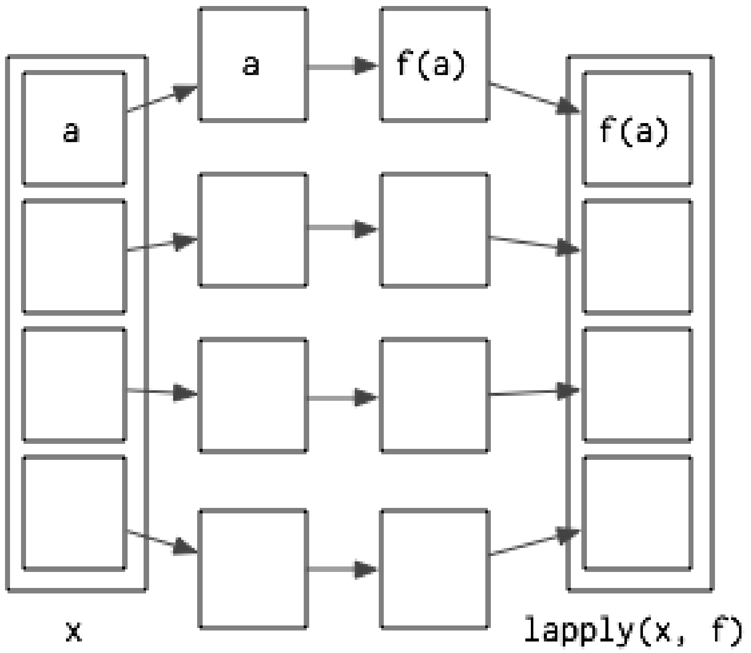


Figure 10.1: A sketch of how `lapply()` works

`lapply()` is written in C for performance, but we can create a simple R implementation that works the same way:

```
lapply2 <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
```

From this code, you can see that `lapply()` is a wrapper around a common for loop pattern: we create a space for output, and then fill it in, applying `f()` to each component of the list. All other for loop functionals build on this base, modifying either the input, the output, or what data the function is applied to. From this code you can see that `lapply()` will also works with vectors: both `length()` and `'[[` work the same way for lists and vectors.

`lapply()` makes it easier to work with lists by eliminating much of the boilerplate, focussing on the operation you're applying to each piece:

```
# Create some random data
l <- replicate(20, runif(sample(1:10, 1)), simplify = FALSE)

# With a for loop
out <- vector("list", length(l))
for (i in seq_along(l)) {
  out[[i]] <- length(l[[i]])
}
unlist(out)
#>  [1]  2  7  2  3  2  7  8 10  8  2  2  3  6 10  5  7  4  4 10  3

# With lapply
unlist(lapply(l, length))
#>  [1]  2  7  2  3  2  7  8 10  8  2  2  3  6 10  5  7  4  4 10  3
```

(We're using `unlist()` to convert the output from a list to a vector to make the output more compact. We'll see other ways of making the output a vector shortly.)

Since data frames are also lists, `lapply()` is useful when you want to do something to each column of a data frame:

```
# What class is each column?
unlist(lapply(mtcars, class))
#>       mpg       cyl      disp        hp      drat        wt      qsec
#> "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
#>        vs        am      gear      carb
#> "numeric" "numeric" "numeric" "numeric"

# Divide each column by the mean
mtcars[] <- lapply(mtcars, function(x) x / mean(x))
```

The pieces of `x` are always supplied as the first argument to `f`. You can override this using R's regular function calling semantics, supplying additional named arguments. For example, imagine you wanted to compute various trimmed means of the same dataset. `trim` is the second parameter of `mean()`, so we want to vary that, keeping the first argument (`x`) fixed. It's easy provided that you remember that the following two calls are equivalent

```
mean(1:100, trim = 0.1)
#> [1] 50.5
mean(0.1, x = 1:100)
#> [1] 50.5
```

So to use `lapply()` with the second argument, we just need to name the first argument:

```
trims <- c(0, 0.1, 0.2, 0.5)
x <- rcauchy(100)
unlist(lapply(trims, mean, x = x))
#> [1] 12.16175 -0.09112 -0.08860  0.01817
```

Looping patterns

When using `lapply()` and friends, it's useful to remember that there are usually three ways to loop over an vector:

1. loop over the elements of the vector: `for(x in xs)`
2. loop over the numeric indices of the vector: `for(i in seq_along(xs))`
3. loop over the names of the vector: `for(nm in names(xs))`

If you're saving the results from a for loop, you usually can't use the first form because it makes very inefficient code. When extending an existing data structure, all the existing data must be copied every time you extend it:

```
xs <- runif(1e3)
res <- c()
for(x in xs) {
  # This is slow!
  res <- c(res, sqrt(x))
}
```

It's much better to create enough space for the output and then fill it in, using the second looping form:

```
res <- numeric(length(xs))
for(i in seq_along(xs)) {
  res[i] <- sqrt(xs[i])
}
```

Corresponding to the three ways to use a for loop there are three ways to use `lapply()` with an object:

```
lapply(xs, function(x) {})
lapply(seq_along(xs), function(i) {})
lapply(names(xs), function(nm) {})
```

Typically you use the first form because `lapply()` takes care of saving the output for you. However, if you need to know the position or the name of the element you're working with, you'll need to use the second or third form; they give you both the position of the object (`i`, `nm`) and its value (`xs[[i]]`, `xs[[nm]]`). If you're struggling to solve a problem using one form, you might find it easier with a different form.

If you're working with a list of functions, remember to use `call_fun`:

```
call_fun <- function(f, ...) f(...)
f <- list(sum, mean, median, sd)
lapply(f, call_fun, x = runif(1e3))
#> [[1]]
#> [1] 498.4
#>
#> [[2]]
#> [1] 0.4984
#>
#> [[3]]
#> [1] 0.4902
#>
#> [[4]]
#> [1] 0.2896
```

Or you could create a variant, `fapply()`, specifically for working with lists of functions:

```
fapply <- function(fs, ...) {
  out <- vector("list", length(fs))
  for (i in seq_along(fs)) {
    out[[i]] <- fs[[i]](...)
  }
  out
}
fapply(f, x = runif(1e3))
#> [[1]]
#> [1] 489.8
#>
#> [[2]]
#> [1] 0.4898
#>
#> [[3]]
#> [1] 0.4802
#>
#> [[4]]
#> [1] 0.2922
```

Exercises

- The function `scale01()` given below scales a vector to have range 0-1. How would you apply it to every column in 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])
  }
  ```

- For each formula in the list below, use a for-loop and lapply to fit the corresponding model to the `mtcars` dataset

  ```
  formulas <- list(
    mpg ~ disp,
    mpg ~ I(1 / disp),
    mpg ~ disp + wt,
    mpg ~ I(1 / disp) + wt
  )
  ```

- Fit the model `mpg ~ disp` to each of the bootstrap replicates of `mtcars` in the list below, using a for loop and then `lapply()`. Can you do it without an anonymous function?

  ```
  bootstraps <- lapply(1:10, function(i) {
    rows <- sample(1:nrow(mtcars), rep = TRUE)
    mtcars[rows, ]
  })
  ```

- For each model in the previous two exercises extract the R^2 using the function below.

  ```
  rsq <- function(mod) summary(mod)$r.squared
  ```

For loop functionals: friends of `lapply()`

The art of using functionals is to recognise what common looping patterns are implemented in existing base functionals, and then use them instead of loops. Once you've mastered the existing functionals, the next step is to start writing your own: if you discover you're duplicating the same looping pattern in many places, you should extract it out into its own function.

The following sections build on `lapply()` and discuss:

- `sapply()` and `vapply()`, variants of `lapply()` that produce vectors, matrices and arrays as **output**, instead of lists.

- `Map()` and `mapply()` which iterate over multiple **input** data structures in parallel.
- **Parallel** versions of `lapply()` and `Map()`, `mclapply()` and `mcMap()`
- **Rolling computations**, showing how a new problem can be solved with for loops, or by building on top of `lapply()`.

Vector output: `sapply` and `vapply`

`sapply()` and `vapply()` are very similar to `lapply()` except they will simplify their output to produce an atomic vector. `sapply()` guesses, while `vapply()` takes an additional argument specifying the output type. `sapply()` is useful for interactive use because it saves typing, but if you use it inside your functions you will get weird errors if you supply the wrong type of input. `vapply()` is more verbose, but gives more informative error messages and never fails silently, so is better suited for use inside other functions.

The following example illustrates these differences. When given a data frame `sapply()` and `vapply()` give the same results. When given an empty list, `sapply()` has no basis to guess the correct type of output, and returns `NULL`, instead of the more correct zero-length logical vector.

```
sapply(mtcars, is.numeric)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
vapply(mtcars, is.numeric, logical(1))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#> TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
sapply(list(), is.numeric)
#> list()
vapply(list(), is.numeric, logical(1))
#> logical(0)
```

If the function returns results of different types or lengths, `sapply()` will silently return a list, while `vapply()` will throw an error. `sapply()` is fine for interactive use because you'll normally notice if something went wrong, but it's dangerous when writing functions.

The following example illustrates a possible problem when extracting the class of columns in data frame: if you falsely assume that class only has one value and use `sapply()` you won't find out about the problem until some future function is given a list instead of a character vector.

```
df <- data.frame(x = 1:10, y = letters[1:10])
sapply(df, class)
```

```
#>         x         y
#> "integer"  "factor"
vapply(df, class, character(1))
#>         x         y
#> "integer"  "factor"

df2 <- data.frame(x = 1:10, y = Sys.time() + 1:10)
sapply(df2, class)
#> $x
#> [1] "integer"
#>
#> $y
#> [1] "POSIXct" "POSIXt"
vapply(df2, class, character(1))
#> Error:     1,
#>   FUN(X[[2]])    2
```

sapply() is a thin wrapper around lapply(), transforming a list into a vector in the final step; vapply() reimplements lapply() but assigns results into a vector (or matrix) of the appropriate type instead of into a list. The following code shows pure R implementation of the essence of sapply() and vapply(); the real functions have better error handling and preserve names, among other things.

```
sapply2 <- function(x, f, ...) {
  res <- lapply2(x, f, ...)
  simplify2array(res)
}

vapply2 <- function(x, f, f.value, ...) {
  out <- matrix(rep(f.value, length(x)), nrow = length(x))
  for (i in seq_along(x)) {
    res <- f(x[i], ...)
    stopifnot(
      length(res) == length(f.value),
      typeof(res) == typeof(f.value)
    )
    out[i, ] <- res
  }
  out
}
vapply2(1:10, is.numeric, logical(1))
#>       [,1]
#>  [1,] TRUE
#>  [2,] TRUE
#>  [3,] TRUE
#>  [4,] TRUE
#>  [5,] TRUE
#>  [6,] TRUE
#>  [7,] TRUE
#>  [8,] TRUE
#>  [9,] TRUE
#> [10,] TRUE
```

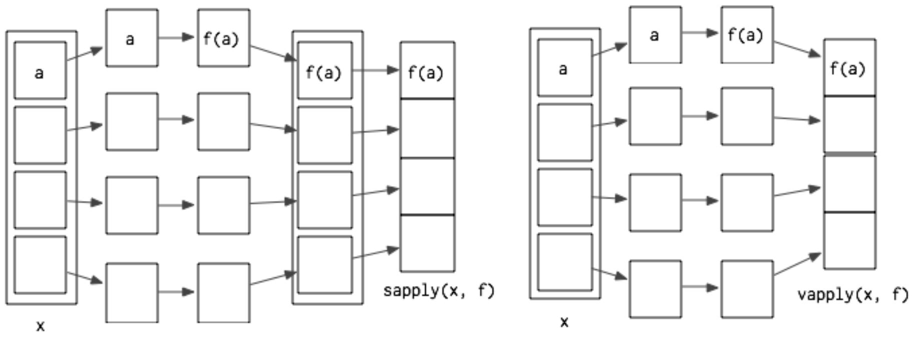


Figure 10.2: Schematics of `sapply` and `vapply`, cf `lapply`.

`vapply()` and `sapply()` are like `lapply()`, but with different outputs; the following section discusses `Map()`, which is like `lapply()` but with different inputs.

Multiple inputs: `Map` (and `mapply`)

With `lapply()`, only one argument to the function varies; the others are fixed. This makes it poorly suited for some problems. For example, how would you find the weighted means when you have two lists, one of observations and the other of weights:

```
# Generate some sample data
xs <- replicate(10, runif(10), simplify = FALSE)
ws <- replicate(10, rpois(10, 5) + 1, simplify = FALSE)
```

It's easy to use `lapply()` to compute the unweighted means:

```
unlist(lapply(xs, mean))
#>  [1] 0.6623 0.6890 0.5477 0.5229 0.5913 0.4282 0.3400 0.5146 0.4785 0.4728
```

But how could we supply the weights to `weighted.mean()`? `lapply(x, means, w)` won't work because the additional arguments to `lapply()` are passed to every call. We could change looping forms:

```
unlist(lapply(seq_along(xs), function(i) {
  weighted.mean(xs[[i]], ws[[i]])
}))
#>  [1] 0.6704 0.6874 0.5362 0.5220 0.6120 0.4238 0.3535 0.4999 0.5025 0.4427
```

This works, but is a little clumsy. A cleaner alternative is to use `Map`, a variant of `lapply()`, where all arguments vary. This lets us write:

```
unlist(Map(weighted.mean, xs, ws))
#>  [1] 0.6704 0.6874 0.5362 0.5220 0.6120 0.4238 0.3535 0.4999 0.5025 0.4427
```

(Note that the order of arguments is a little different: with `Map()` the function is the first argument, with `lapply()` it's the second.

This is equivalent to:

```
stopifnot(length(x) == length(w))
out <- vector("list", length(x))
for (i in seq_along(x)) {
  out[[i]] <- weighted.mean(x[[i]], w[[i]])
}
```

There's a natural equivalence between `Map()` and `lapply()` because you can always convert a `Map()` to an `lapply()` that iterates over indices, but using `Map()` is more concise, and more clearly indicates what you're trying to do.

`Map` is useful whenever you have two (or more) lists (or data frames) that you need to process in parallel. For example, another way of standardising columns, is to first compute the means and then divide by them. We could do this with `lapply()`, but if we do it in two steps, we can more easily check the results at each step, which is particularly important if the first step is more complicated.

```
mtmeans <- lapply(mtcars, mean)
mtmeans[] <- Map(`/`, mtcars, mtmeans)

# In this case, equivalent to
mtcars[] <- lapply(mtcars, function(x) x / mean(x))
```

If some of the arguments should be fixed, and not varying, you need to use an anonymous function:

```
Map(function(x, w) weighted.mean(x, w, na.rm = TRUE), xs, ws)
```

We'll see a more compact way to express the same idea in the next chapter.

You may be more familiar with `mapply()` than `Map()`. I prefer `Map()` because:

- it is equivalent to `mapply` with `simplify = FALSE`, which is almost always what you want.
- Instead of using an anonymous function to provide constant inputs, `mapply` has the `MoreArgs` argument which takes a list of extra arguments that will be supplied, as is, to each call. This breaks R's usual lazy evaluation semantics, and is inconsistent with other functions.

In brief, `mapply()` is more complicated for little gain.

Rolling computations

What if you need a for-loop replacement that doesn't exist in base R? You can often create your own by recognising common looping structures and implementing your own wrapper. For example, you might be interested in smoothing your data using a rolling (or running) mean function:

```r
rollmean <- function(x, n) {
  out <- rep(NA, length(x))

  offset <- trunc(n / 2)
  for (i in (offset + 1):(length(x) - n + offset - 1)) {
    out[i] <- mean(x[(i - offset):(i + offset - 1)])
  }
  out
}
x <- seq(1, 3, length = 1e2) + runif(1e2)
plot(x)
lines(rollmean(x, 5), col = "blue", lwd = 2)
lines(rollmean(x, 10), col = "red", lwd = 2)
```

But if the noise was more variable (i.e. it had a longer tail) you might worry that your rolling mean was too sensitive to the occasional outlier and instead implement a rolling median.

```r
x <- seq(1, 3, length = 1e2) + rt(1e2, df = 2) / 3
plot(x)
lines(rollmean(x, 5), col = "red", lwd = 2)
```

To modify `rollmean()` to `rollmedian()` all you need to do is replace `mean` with `median` inside the loop, but instead of copying and pasting to create a new function, we could extract the idea of computing a rolling summary into its own function:

```
rollapply <- function(x, n, f, ...) {
  out <- rep(NA, length(x))

  offset <- trunc(n / 2)
  for (i in (offset + 1):(length(x) - n + offset - 1)) {
    out[i] <- f(x[(i - offset):(i + offset - 1)], ...)
  }
  out
}
plot(x)
lines(rollapply(x, 5, median), col = "red", lwd = 2)
```

You might notice that the internal loop looks pretty similar to a `vapply()` loop, so we could rewrite the function as:

```
rollapply <- function(x, n, f, ...) {
  offset <- trunc(n / 2)
  locs <- (offset + 1):(length(x) - n + offset - 1)
  vapply(locs, function(i) f(x[(i - offset):(i + offset - 1)], ...),
    numeric(1))
}
```

This is effectively the same as the implementation in `zoo::rollapply()`, but it provides many more features and much more error checking.

Parallelisation

One thing that's interesting about the defintions of `lapply()` is that because each iteration is isolated from all others, the order in which they are computed doesn't matter. For example, while `lapply3()`, defined below, scrambles the order in which computation occurs, the results are same every time:

```
lapply3 <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in sample(seq_along(x))) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
unlist(lapply3(1:10, sqrt))
#>  [1] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000 3.162
unlist(lapply3(1:10, sqrt))
#>  [1] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000 3.162
```

This has a very important consequence: since we can compute each element in any order, it's easy to dispatch the tasks to different cores, and compute in parallel. This is what `mclapply()` (and `mcMap`) in the parallel package do:

```
library(parallel)
unlist(mclapply(1:10, sqrt, mc.cores = 4))
#>  [1] 1.000 1.414 1.732 2.000 2.236 2.449 2.646 2.828 3.000 3.162
```

In this case `mclapply()` is actually slower than `lapply()`, because the cost of the individual computations is low, and some additional work is needed to send the computation to the different cores then collect the results together. If we take a more realistic example, generating bootstrap replicates of a linear model, we see more of an advantage:

```
boot_df <- function(x) x[sample(nrow(x), rep = T), ]
rsquared <- function(mod) summary(mod)$r.square
boot_lm <- function(i) {
  rsquared(lm(mpg ~ wt + disp, data = boot_df(mtcars)))
}

system.time(lapply(1:500, boot_lm))
#>
#> 1.492 0.024 1.527
system.time(mclapply(1:500, boot_lm, mc.cores = 2))
#>
#> 0.704 0.100 1.056
```

It is rare to get an exactly linear improvement with increasing number of cores, but if your code uses `lapply()` or `Map()`, this is an easy way to improve performance.

Exercises

- Use `vapply()` to:
- Compute the standard deviation of every column in a numeric data frame.
- Compute the standard deviation of of every numeric column in a mixed data frame (Hint: you'll need to use `vapply()` twice)
- Recall: why is using `sapply()` to get the `class()` of each element in a data frame dangerous?
- The following code simulates the performance of a t-test for non-normal data. Use `sapply()` and an anonymous function to extract the p value from every trial. Extra challenge: get rid of the anonymous function and use the `'[[` function.

```
trials <- replicate(100, t.test(rpois(10, 10), rpois(7, 10)),
  simplify = FALSE)
```

- Implement a combination of `Map()` and `vapply()` to create an `lapply()` variant that iterates in parallel over all of its inputs and stores its outputs in a vector (or a matrix). What arguments should the function take?
- What does `replicate()` do? What sort of for loop does it eliminate? Why do its arguments differ from `lapply()` and friends?
- Implement `mcsapply()`, a multicore version of `sapply()`. Can you implement `mcvapply()` a parallel version of `vapply()`? Why/why not?
- Implement a version of `lapply()` that supplies `f()` with both the name and the value of each component.

Data structure functionals

As well as functionals that exist to eliminate common looping constructs, another family of functionals works to eliminate loops for common data manipulation tasks. In this section, we'll give a brief overview of the available options. We'll show you some of the available options, hint at how they can help you, and point you in the right direction to learn more. We'll cover three categories of data structure functionals:

- base functions for working with matrices: `apply()`, `sweep()` and `outer()`
- `tapply()`, which summarises a vector divided into groups by the values of another vector
- the `plyr` package, which generalises the ideas of `tapply()` to work with inputs of data frames, lists and arrays, and outputs of data frames, lists, arrays and nothing

Matrix and array operations

So far, all the functionals we've seen work with 1d input structures. The three functionals in this section provide useful tools for working with high-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, 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 does not have a simplify argument, so you can never be completely sure what type of output you will get. This generally 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()` is a function that allows you to "sweep" out the values of a summary statistic. It is most often useful in conjunction with `apply()` and it often used to standardise arrays in some way. 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(1:9, 1:9, "*")
#>       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
#>  [1,]    1    2    3    4    5    6    7    8    9
#>  [2,]    2    4    6    8   10   12   14   16   18
#>  [3,]    3    6    9   12   15   18   21   24   27
#>  [4,]    4    8   12   16   20   24   28   32   36
#>  [5,]    5   10   15   20   25   30   35   40   45
#>  [6,]    6   12   18   24   30   36   42   48   54
#>  [7,]    7   14   21   28   35   42   49   56   63
#>  [8,]    8   16   24   32   40   48   56   64   72
#>  [9,]    9   18   27   36   45   54   63   72   81
```

Good places to learn more about `apply()` and friends are:

- "Using apply, sapply, lapply in R"[2] by Peter Werner.
- "The infamous apply function"[3] by Slawa Rokicki.
- "The R apply function – a tutorial with examples"[4] by axiomOfChoice.
- The stack overflow question "R Grouping functions: sapply vs. lapply vs. apply. vs. tapply vs. by vs. aggregate vs"[5].

Group apply

You can think about `tapply()` as a generalisation to `apply()` that allows for "ragged" arrays, where each row can have different numbers of columns. This is often needed when you're trying to summarise a data set. For example, imagine you've collected some pulse rate 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
#> 70.50 74.67
```

[2]http://petewerner.blogspot.com/2012/12/using-apply-sapply-lapply-in-r.html
[3]http://rforpublichealth.blogspot.no/2012/09/the-infamous-apply-function.html
[4]http://forgetfulfunctor.blogspot.com/2011/07/r-apply-function-tutorial-with-examples.html
[5]http://stackoverflow.com/questions/3505701

It's easiest to understand how `tapply()` works by first creating a "ragged" data structure from the inputs. This is the job of the `split()` function, which takes two inputs and returns a list, where all the elements in the first vector with equal entries in the second vector get put in the same element of the list:

```
split(pulse, group)
#> $A
#>  [1] 70 66 69 71 63 74 73 73 75 71
#>
#> $B
#>  [1] 75 76 76 75 78 75 75 74 73 75 78 66
```

Then you can see that `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
#> 70.50 74.67
```

Be able to rewrite our `tapply()` as a combination of `split()` and `sapply()` is a good indication that we've been able to extract independent pieces that we can recombine to solve new problems.

The plyr package

One challenge with using the base functionals is that they have grown organically over time, and have been written by multiple authors. This means that they are not very consistent. For example,

- The simplify argument is called `simplify` in `tapply()` and `sapply()`, but `SIMPLIFY` for `mapply()`, and `apply()` lacks the argument altogether.
- `vapply()` is a variant of `sapply()` that allows you to describe what the output should be, but there are no corresponding variants of `tapply()`, `apply()`, or `Map()`.
- The first argument to most functionals is the vector, but the first argument to `Map()` is the function.

This makes learning these operators challenging, as you have to memorise all of the variations. Additionally, if you think about the combination of input and output types, base R only provides a partial set of functions:

	list	data frame	array
list	lapply		sapply
data frame	by		
array			apply

This was one of the driving forces behind the creation of the plyr package, which provides consistently named functions with consistently named arguments and implements all combinations of input and output data structures:

	list	data frame	array
list	llply	ldply	laply
data frame	dlply	ddply	daply
array	alply	adply	aaply

Each of these functions splits up the input, applies a function to each piece and then joins the results back together. Overall, this process is called "split-apply-combine", and you can read more about it and plyr in The Split-Apply-Combine Strategy for Data Analysis[6], an open-access article published in the Journal of Statistical Software.

Exercises

- How does `apply()` arrange the output. Read the documentation and perform some experiments.
- There's no equivalent to `split()` + `vapply()`. Should there be? When would it be useful? Implement it yourself.
- Implement a pure R version of `split()`. (Hint: use unique and subseting)
- What other types of input and output are missing? Brainstorm before you look up some answers in the plyr paper[7]

[6] http://www.jstatsoft.org/v40/i01/
[7] http://www.jstatsoft.org/v40/i01/

Functional programming

Another way of thinking about functionals is as a set of general tools for altering, subsetting and collapsing lists. Every functional programming has three tools for this: `Map()`, `Reduce()`, and `Filter()`. We've seen `Map()` already, and the following sections describe `Reduce()`, a powerful tool for extending two-argument functions, and `Filter()`, a member of an important class of functionals that work with predicates, functions that return a single boolean.

`Reduce()`

`Reduce()` recursively reduces a vector, `x`, to a single value by recursively calling a function `f` with 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. Reduce is also known as fold, because it folds together adjacent elements in the list.

The following two examples show what `Reduce` does with an infix and prefix function:

```
Reduce(`+`, 1:3)
((1 + 2) + 3)

Reduce(sum, 1:3)
sum(sum(1, 2), 3)
```

As you might have come to expect by now, the essence of `Reduce()` can be described by a simple for loop:

```
Reduce2 <- function(f, x) {
  out <- x[[1]]
  for(i in seq(2, length(x))) {
    out <- f(out, x[[i]])
  }
  out
}
```

The real `Reduce()` is more complicated because it includes arguments to control whether the values are reduced from the left or from the right (`right`), an optional initial value (`init`), and an option to output every intermediate result (`accumulate`).

Reduce is an elegant way of turning binary functions into functions that can deal with any number of arguments. It's useful for implementing many types of recursive operations, like merges and intersections (We'll see another use in the final case study). For example, imagine you had a list of numeric vectors, and you wanted to find the values that occurred in every element:

```
l <- replicate(5, sample(1:10, 15, rep = T), simplify = FALSE)
l
#> [[1]]
#>  [1]  5  8  4  3  5  4  3  6  5  3  7  3  8 10 10
#> 
#> [[2]]
#>  [1] 2 2 3 4 9 3 5 6 5 2 5 9 7 4 9
#> 
#> [[3]]
#>  [1]  1  8  6  9  8  7  8  4 10 10  9  4 10  7  1
#> 
#> [[4]]
#>  [1]  1  1  1  5  2  6  8 10  5  9  4  1  3  1  9
#> 
#> [[5]]
#>  [1]  2  8  3  4  1  1  8  3  1  2  8  4  4  5 10
```

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] 4
```

That's hard to read because of the dagwood sandwich problem, and is equivalent to:

```
Reduce(intersect, l)
#> [1] 4
```

Predicate functionals

A **predicate** is a function that returns a single `TRUE` or `FALSE`, like `is.character`, `all`, or `is.NULL`. `is.na` isn't a predicate function because it returns a vector of values. Predicate functionals make it easy to apply predicates to lists or data frames. There are a three useful predicate functionals in base R: `Filter()`, `Find()` and `Position()`.

- `Filter`: returns a new vector containing only elements where the predicate is `TRUE`.

- `Find()`: return the first element that matches the predicate (or the last element if `right = TRUE`).
- `Position()`: return the position of the first element that matches the predicate (or the last element if `right = TRUE`).

Another useful functional makes it easy to generate a logical vector from a list (or a data frame) and a predicate:

```
where <- function(f, x) {
  vapply(x, f, logical(1))
}
```

The following example shows how you might use these functionals with a data frame:

```
str(Filter(is.factor, iris))
where(iris, is.factor)
str(Find(is.factor, iris))
Position(is.factor, iris)
```

One function I use a lot is `compact()`:

```
compact <- function(x) Filter(function(y) !is.null(y), x)
```

It removes all null elements from a list - you'll see it again in the [[function operators]] chapter.

Exercises

- Use `Filter()` and `vapply()` to create a function that applies a summary statistic to every column in a data frame.
- What's the relationship between `which()` and `Position()`?
- Re-write `compact` to eliminate the anonymous function.
- 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 the complementary `All` function.
- Implement the `span` function from Haskell, which given a list `x` and a predicate function `f`, returns the longest sequential run of elements where the predicate is true. (Hint: you might find `rle()` helpful. Make sure to read the source and figure out how it works)

Mathematical functionals

Functionals are very common in mathematics. The limit, the maximum, the roots (the set of points where `f(x) = 0`), and the definite integral are all functionals: given a function, they return a single number (or a 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 see all of them are implemented using an algorithm that involves iteration.

In this section we'll explore some of R's built-in mathematical functionals. There are three functions that work with functions that return a single numeric value:

- `integrate`: find the area under the curve given by `f`
- `uniroot`: find where `f` hits zero
- `optimise`: find location of lowest (or highest) value of `f`

Let's explore how these are used with a simple function, `sin`:

```r
integrate(sin, 0, pi)
#> 2 with absolute error < 2.2e-14
uniroot(sin, pi * c(1 / 2, 3 / 2))
#> $root
#> [1] 3.142
#>
#> $f.root
#> [1] 1.225e-16
#>
#> $iter
#> [1] 2
#>
#> $estim.prec
#> [1] 6.104e-05
optimise(sin, c(0, 2 * pi))
#> $minimum
#> [1] 4.712
#>
#> $objective
#> [1] -1
optimise(sin, c(0, pi), maximum = TRUE)
#> $maximum
#> [1] 1.571
#>
#> $objective
#> [1] 1
```

In statistics, optimisation is often used for maximum likelihood estimation. Maximum likelihood estimation (MLE) is a natural fit for functional programming because we have a well defined problem domain and a general technique to solve it. In MLE, we have two sets of parameters: the data, which is fixed for a given problem, and the parameters, which will vary as we try to find the maximum. That fits naturally with closures because we can have two layers of parameters to a closure. Closures plus optimisation gives rise to an approach to solving MLE problems like the following.

First, we create a function that computes the negative log likelihood (NLL) for a given dataset. In R, it's common to use the negative since `optimise()` defaults to finding the minimum.

```
poisson_nll <- function(x) {
  n <- length(x)
  function(lambda) {
    n * lambda - sum(x) * log(lambda) # + terms not involving lambda
  }
}
```

With the general NLL in hand, we create two specific NLL functions for two datasets, and use `optimise()` to find the best values, given a generous starting range.

```
nll1 <- poisson_nll(c(41, 30, 31, 38, 29, 24, 30, 29, 31, 38))
nll2 <- poisson_nll(c(6, 4, 7, 3, 3, 7, 5, 2, 2, 7, 5, 4, 12, 6, 9))

optimise(nll1, c(0, 100))$minimum
#> [1] 32.1
optimise(nll2, c(0, 100))$minimum
#> [1] 5.467
```

We can verify these values are correct by using the analytic solution: in this case, it's just the mean of the data values, 32.1 and 5.45.

Another important mathematical functional is `optim()`. It is a generalisation of `optimise()` to more than one dimension. If you're interested in how `optim()` works, you might want to explore the `Rvmmin` package, which provides a pure-R implementation of R. Interestingly `Rvmmin` is no slower than `optim()`, even though it is written in R, not C: for this problem, the bottleneck is evaluating the function multiple times, not controlling the optimisation.

Exercises

- Implement the `arg_max` function. It should take a function, and a vector of inputs, returning the elements of the input where the function returns the highest number. 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`.

- Challenge: read about the fixed point algorithm[8]. Complete the exercises using R.

Converting loops to functionals, and when it's not possible

There are a wide class of for loops that do not naturally match of the functionals we've described so far. Sometimes it's possible to torture your code to make it work, but it's usually not a good idea: for loops are verbose and not very expressive, but all R programmers are familiar with them. It takes a while before you can identify whether or not a for loop has a related vectorised solution, or a matching functional. It's also easy to go too far: trying to convert for loops that really should stay as loops. This section provides some more resources for learning and highlights three types of loop that you shouldn't try and convert into a functional:

- modifying in place
- recursive functions
- while loops

Stackoverflow is a good resource for learning more about converting for loops to use functionals. A couple of questions and answers that I think are particularly helpful are:

- "Alternative to loops in R"[9]
- "Speed up the loop operation in R"[10]

You can also look for other similar questions that have cropped up since I wrote this with a search[11]

Modifying in place

If you need to modify part of an existing data frame, it's often better to use a for loop. For example, the following code sample performs a variable-by-variable transformation by matching the names of a list of functions to the names of variables in a data frame.

[8]http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-12.html#%_sec_1.3
[9]http://stackoverflow.com/a/14520342/16632
[10]http://stackoverflow.com/a/2970284/16632
[11]http://stackoverflow.com/search?tab=votes&q=%5br%5d%20for%20loop

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) factor(x, levels = c("auto", "manual"))
)
for(var in names(trans)) {
  mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

We couldn't normally use `lapply()` to replace this loop directly, but it is *possible* to replace the loop with `lapply()` by using `<<-`:

```
lapply(names(trans), function(var) {
  mtcars[[var]] <<- trans[[var]](mtcars[[var]])
})
```

We've eliminated the for loop, but our code is longer and we've had to use an unusual language feature, `<<-`. And to understand what `mtcars[[var]] <<- ...` does, you have to understand not only how `<<-` works, but also what `x[[y]] <<- z` does behind the scenes. We've taken a simple, easily understood for loop, and turned it into something few people will understand: not a good idea!

Recursive relationships

Another case where it's hard to convert a for loop into a functional is when the relationship is defined recursively. For example, exponential smoothing smoothes data values by taking a weighted average of the current and previous point. The `exps()` function below implements exponential smoothing with a for loop.

```
exps <- function(x, alpha) {
  s <- numeric(length(x) + 1)
  for (i in seq_along(s)) {
    if (i == 1) {
      s[i] <- x[i]
    } else {
      s[i] <- alpha * x[i - 1] + (1 - alpha) * s[i - 1]
    }
  }
  s
}
x <- runif(10)
exps(x, 0.5)
#>  [1] 0.8535 0.8535 0.4356 0.6452 0.4794 0.5209 0.5795 0.5397 0.6378 0.7732
#> [11] 0.4325
```

We can't eliminate the for loop because none of the functionals we've seen allow the output at position `i` to depend on the input and output at position `i - 1`.

If we encountered this pattern a lot, we could create our own function. I've called it `lbapply()`, where `lb` is short for lookback.

```
lbapply <- function(x, f, init = x[1], ...) {
  out <- numeric(length(x))
  out[1] <- init
  for(i in seq(2, length(x))) {
    out[i] <- f(x[i - 1], out[i - 1], ...)
  }
  out
}

f <- function(x, out, alpha) alpha * x + (1 - alpha) * out
lbapply(x, f, alpha = 0.5)
#>  [1] 0.8535 0.8535 0.4356 0.6452 0.4794 0.5209 0.5795 0.5397 0.6378 0.7732
```

This is only worthwhile if we need this function frequently: otherwise it just places an additional cognitive burden on the reader.

Another solution for eliminate the for loop in these cases is to solve the recurrence relation[12], removing the recursion and replacing it with explicit references. This requires a new set of tools, and is mathematically challenging, but it can pay off by producing a simpler function. For exponential smoothing, it is possible to rewrite in terms of `i`:

```
exps1 <- function(x, alpha) {
  n <- length(x)
  i <- seq_along(x) - 1
  cumsum(alpha * rev(x[-1]) * (1 - alpha) ^ i[-n]) +
    (1 - alpha) ^ (n - 1) * x[1]
}
exps1(x, 0.5)
#> [1] 0.04759 0.27473 0.36672 0.39797 0.41791 0.42670 0.42915 0.43248 0.43252
```

It's arguable whether or not that's more understandable for this case, but it may be useful for your problem. We'll see another example of a function defined recursively, the Fibonacci series, in the [[SoftwareSystems]] chapter.

While loops

Another type of looping construct in R is the `while` loop: this keeps running code until a condition is met. `while` loops are more general than `for` loops

[12] http://en.wikipedia.org/wiki/Recurrence_relation#Solving

because you can rewrite every for loop as a while loop, but you can't do the opposite. For example, this for loop:

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

Can be turned into this while loop:

```
i <- 1
while(i <= 10) {
  print(i)
  i <- i + 1
}
```

Not every while loop can be turned into a for loop, because many while loops don't know in advance how many times they will be run:

```
i <- 0
while(TRUE) {
  if (runif(1) > 0.9) break
  i <- i + 1
}
```

This is a common situation when you're writing simulations: one of the random parameters in your simulation may be how many times a process occurs.

In some cases, like above, you may be able to remove the loop by recongnising some special feature of the problem. For example, the above problem is counting how many times a Bernoulli trial with $p = 0.1$ is run before it is successful: this is a geometric random variable so you could replace the above code with `i <- rgeom(1, 0.1)`. Similar to solving recurrence relations, this is extremely difficult to do in general, but you'll get big gains if you can do it for your situation.

A family of functions

The following case study shows how you can use functionals to start small, with very simple functions, then build them up into more complicated and featureful tools. We'll start with a simple idea, adding two numbers together, and show how we can extend it to summing multiple numbers, computing parallel sums, cumulative sums, and sums for arrays. While we'll illustrate the ideas with addition, and you can use exactly the same ideas for multiplication, smallest and largest, and string concatenation to generate a wide family of functions, including over 20 functions provided in base R.

We'll start by defining a very simple plus function, that takes two scalar arguments:

```
add <- function(x, y) {
  stopifnot(length(x) == 1, length(y) == 1,
    is.numeric(x), is.numeric(y))
  x + y
}
```

(We're using R's existing addition operator here, which does much more, but the focus in this section is on how we can take very very simple functions and extend them to do more).

We really should also have some way to deal with missing values. A helper function will make this a bit easier: if `x` is missing it should return `y`, if `y` is missing it should returns `x`, and if both `x` and `y` are missing then it should returns another argument to the function: `identity`. (We'll talk a bit later about while we've called it identity). This function is probably a bit more general than what we need now, but it will come in handy when you implement other binary operators.

```
rm_na <- function(x, y, identity) {
  if (is.na(x) && is.na(y)) {
    identity
  } else if (is.na(x)) {
    y
  } else {
    x
  }
}
rm_na(NA, 10, 0)
#> [1] 10
rm_na(10, NA, 0)
#> [1] 10
rm_na(NA, NA, 0)
#> [1] 0
```

That allows us to write a version of `add` that can deal with missing values if needed: (and it often is!)

```
add <- function(x, y, na.rm = FALSE) {
  if (na.rm && (is.na(x) || is.na(y))) rm_na(x, y, 0) else x + y
}
add(10, NA)
#> [1] NA
add(10, NA, na.rm = TRUE)
#> [1] 10
add(NA, NA)
```

```
#> [1] NA
add(NA, NA, na.rm = TRUE)
#> [1] 0
```

Why did we pick an identity of `0`? Why should `add(NA, NA, na.rm = TRUE)` return 0? Well, for every other input it returns a numeric vector of length 1, so it should do that even if both arguments are missing values. We next need to figure out what that number should be. We can use a special property of add to work this out: add is associative, which the order of addition doesn't matter. In other words, the following two function calls should return the same value:

```
add(add(3, NA, na.rm = TRUE), NA, na.rm = TRUE)
#> [1] 3
add(3, add(NA, NA, na.rm = TRUE), na.rm = TRUE)
#> [1] 3
```

That implies that `add(NA, NA, na.rm = TRUE)` must be 0.

Now we have the basics working, we can extend this function to deal with more complicated inputs. The first way we might want to extend it is add more than two numbers together. This is a simple application of `Reduce`: if the input is `c(1, 2, 3)`, then we want to compute `add(1, add(2, 3))`:

```
r_add <- function(xs, na.rm = TRUE) {
  Reduce(function(x, y) add(x, y, na.rm = na.rm), xs)
}
r_add(c(1, 4, 10))
#> [1] 15
```

This looks good, but we need to test it for a few special cases:

```
r_add(NA, na.rm = TRUE)
#> [1] NA
r_add(numeric())
#> NULL
```

These are incorrect: in the first case we get a missing value even thought we've explicitly asked for them to be ignored, and in the second case we get `NULL`, instead of a length 1 numeric vector (as for every other set of inputs).

The two problems are related: if we give `Reduce()` a length one vector it doesn't have anything to reduce, so it just returns the input; if we give it a length 0 input it always returns `NULL`. There are two ways to fix this: we can concatenate `0` to every input vector, or we can use the `init` argument to `Reduce()` (which effectively does the same thing):

```
r_add <- function(xs, na.rm = TRUE) {
  Reduce(function(x, y) add(x, y, na.rm = na.rm), c(0, xs))
}
r_add(c(1, 4, 10))
#> [1] 15
r_add(NA, na.rm = TRUE)
#> [1] 0
r_add(numeric())
#> [1] 0
```

(This is equivalent to `sum()`)

It would also be nice to have a vectorised version of `add` so that we can give it two vectors of numbers to add in parallel. We have two ways, using `Map()` or `vapply()`, to implement this, neither of which are perfect. `Map()` returns a list (we want a numeric vector), and while `vapply()` returns a vector, we'll need to loop over the indices.

A few test cases makes sure that it behaves as we expect. We're a bit stricter than base R here because we don't do recycling - you could add that if you wanted, but I find problems with recycling a common source of silent bugs.

```
v_add <- function(x, y, na.rm = TRUE) {
  stopifnot(length(x) == length(y), is.numeric(x), is.numeric(y))
  Map(function(x, y) add(x, y, na.rm = na.rm), x, y)
}

v_add <- function(x, y, na.rm = TRUE) {
  stopifnot(length(x) == length(y), is.numeric(x), is.numeric(y))
  vapply(seq_along(x), function(i) add(x[i], y[i], na.rm = na.rm),
    numeric(1))
}
v_add(1:10, 1:10)
#>  [1]  2  4  6  8 10 12 14 16 18 20
v_add(numeric(), numeric())
#> numeric(0)
v_add(c(1, NA), c(1, NA))
#> [1] 2 0
v_add(c(1, NA), c(1, NA), na.rm = TRUE)
#> [1] 2 0
```

(This is the usual behavior of `+` in R, although we have more control over missing values.)

Another variant of adding is the cumulative sum: it's like the reductive version, but we see every step along the way to the final result. This is easy to implement with `Reduce()`'s `accumuate` argument:

```
c_add <- function(xs, na.rm = FALSE) {
  Reduce(function(x, y) add(x, y, na.rm = na.rm), xs,
    accumulate = TRUE)
}
c_add(1:10)
#>  [1]  1  3  6 10 15 21 28 36 45 55
c_add(10:1)
#>  [1] 10 19 27 34 40 45 49 52 54 55
```

(This function is equivalent to `cumsum()`)

Finally, we might want to define versions for more complicated data structures like matrices. We could create `row` and `col` variants that sum across rows and columns respectively, or we could go the whole hog and define an array version that would sum across any arbitrary dimensions of an array. These are easy to implement: they're a combination of `add()` and `apply()`

```
row_sum <- function(x, na.rm = TRUE) apply(x, 1, add, na.rm = na.rm)
col_sum <- function(x, na.rm = TRUE) apply(x, 2, add, na.rm = na.rm)
arr_sum <- function(x, dim, na.rm = TRUE) apply(x, dim, add, na.rm = na.rm)
```

(These are equivalent to `rowSums()` and `colSums()`)

If every function we have created already has an existing equivalent in base R, why did we bother? There are two main reasons:

- we've created all our variants from a very simple binary operator (`add`) and a well-tested functional (`Reduce`, `Map` and `apply`), so we know all the variants will behave consistently.
- we've seen the infrastructure for addition, so we can now adapt it to other operators that might not have the full suite variants in base R.

The downside of this approach is that these implementations are unlikely to be efficient (For example, `colSums(x)` is much faster than `apply(x, 2, sum)`). However, even if they don't turn out to be fast enough, they are still a good starting point because they are less likely to have bugs; and when you create faster versions (maybe using [[Rcpp]]), you can compare results to make sure your fast versions are still correct.

If you enjoyed this section, you might also enjoy List out of lambda[13], a blog article by Steve Losh that explores how you can produces higher level language structures (like lists) out of more primitive language features (like closures, aka lambdas).

[13]http://stevelosh.com/blog/2013/03/list-out-of-lambda/

Exercises

- Implement `smaller` and `larger` functions that given two inputs return either the smaller or the larger value. Implement `na.rm = TRUE`: what should the identity be? (Hint: `smaller(x, smaller(NA, NA, na.rm = TRUE), na.rm = TRUE)` must be `x`, so `smaller(NA, NA, na.rm = TRUE)` must be bigger than any other value of x.). Use `smaller` and `larger` to implement equivalents of `min`, `max`, `pmin`, `pmax`, and new functions `row_min` and `row_max`
- Create a table that has add, multiply, smaller, larger, and, and or in the columns and binary operator, reducing variant, vectorised variant, array variants in the rows.
- Fill in the cells with the names of base R functions that perform each of the roles
- Compare the names and arguments of the existing R functions. How consistent are they? How could you improve them?
- Complete the matrix by implementing any missing functions
- How does `paste()` fit into this structure? What is the scalar binary function that underlies `paste()`? What are the `sep` and `collapse` arguments to `paste()` equivalent to? Are there are any paste variants that don't have existing R implementations?

Chapter 11

Function operators

In this chapter, you'll learn about function operators: functions that take one (or more) functions as input and return a function as output. Function operators are a FP technique related to functionals, but where functionals abstract away common uses of loops, function operators abstract over common uses of anonymous functions. Like functionals, there's nothing you can't do without them; but they can make your code more readable, more expressive and faster to write.

Here's an example of a simple function operator (FO) that makes a function chatty, showing its input and output (albeit in a naive way). It's useful because it gives a window into functionals, and we can use it to see how `lapply()` and `mclapply()` execute code differently. (We'll explore this theme in more detail below with the fully-featured `tee()` function)

```
library(parallel)
chatty <- function(f) {
  function(x) {
    res <- f(x)
    cat(format(x), " -> ", format(res, digits = 3), "\n", sep = "")
    res
  }
}
s <- c(0.4, 0.3, 0.2, 0.1)
x2 <- lapply(s, chatty(Sys.sleep))
x2 <- mclapply(s, chatty(Sys.sleep))

# 0.3 -> NULL
# 0.4 -> NULL
# 0.1 -> NULL
# 0.2 -> NULL
```

In the last chapter, we saw that most built-in functionals, like `Reduce`, `Filter` and `Map`, have very few arguments, and we used anonymous functions to modify how they worked. In this chapter, we'll start to build up tools that replace standard anonymous functions with specialised equivalents that allow us to communicate our intent more clearly. For example, in the last chapter we used an anonymous function plus `Map` to supply fixed arguments:

```
Map(function(x, y) f(x, y, zs), xs, ys)
```

Later in this chapter, we'll learn about partial application and the `partial()` function. Partial application encapsulates the use of an anonymous function to supply default arguments, and leads to the following succinct code:

```
Map(partial(f, zs = zs), xs, yz)
```

This is an important use of FOs: you can eliminate parameters to a functional by instead transforming the input function. This approach allows your functionals to be more extensible: as long as the inputs and outputs of the function remain the same, your functional can be extended in ways you haven't thought of.

In this chapter, we'll explore four types of function operators (FOs). Function operators can:

- **add behaviour** while leaving the function otherwise unchanged, like automatically logging when the function is run, ensuring a function is run only once, or delaying the operation of a function.
- **change output**, for example by returning a value if the function throws an error, or negating the result of a logical predicate.
- **change input**, like partially evaluating the function, converting a function that takes multiple arguments to a function that takes a list, or automatically vectorising a function.
- **combine functions**, for example, combining the results of predicate functions with boolean operators, or composing multiple function calls.

For each type, we'll show you useful FOs, and how you can use them as another way of describing tasks in R: as combinations of multiple functions instead of combinations of arguments to a single function. The goal is not to provide an exhaustive list of every possible FO, but to show a selection and demonstrate how well with each other and in concert with functionals. For your own work, you will need to think about and experiment with what function operators help you solve recurring problems.

The examples in this chapter come from five years of creating function operators in different R packages (particularly plyr), and from reading about useful operators in other languages.

In other languages

Function operators are used extensively in FP languages like Haskell, and are common in Lisp, Scheme and Clojure. They are an important part of modern JavaScript programming, like in the underscore.js[1] library, and are particularly common in CoffeeScript, since the syntax for anonymous functions is so concise. Stack based languages like Forth and Factor use function operators almost exclusively, since it is rare to refer to variables by name. Python's decorators are just function operators by a different name[2]. They are very rare in Java, because it's difficult to manipulate functions (although possible if you wrap them up in strategy-type objects), and also rare in C++; while it's possible to create objects that work like functions ("functors") by overloading the `()` operator, modifying these objects with other functions is not a common programming technique. That said, C++ 11 adds partial application (`std::bind`) to the standard library.

Behavioural FOs

The first class of FOs are those that leave the inputs and outputs of a function unchanged, but add some extra behaviour. In this section, we'll see functions that:

- log to disk everytime a function is run
- add a delay to avoid swamping a server with work
- print to console every n invocations (useful if you want to check on a long running process)
- save time by caching previous computations

To make these use cases concrete, imagine we want to download a long vector of urls with `download.file()`. That's pretty simple with `lapply()`:

```
lapply(urls, download.file, quiet = TRUE)
```

(This example ignores the fact that `download.file` also needs a file name, so pretend it has a useful default for the purposes of this exposition.)

Because we have a long list we want to print some output so that we know it's working (we'll print a `.` every ten urls), and we also want to avoid hammering the server, so we add a small delay to the function between each call. That leads to a rather more complicated for loop (we can no longer use `lapply()` because we need an external counter):

[1] http://underscorejs.org/
[2] http://stackoverflow.com/questions/739654/

```
i <- 1
for(url in urls) {
  i <- i + 1
  if (i %% 10 == 0) cat(".")
  Sys.delay(1)
  download.file(url, quiet = TRUE)
}
```

Reading this code is quite hard because we are using low-level functions, and it's not obvious (without some thought), what the overall objective is. In the remainder of this chapter we'll create FOs that encapsulate each of the modifications, allowing us to write:

```
lapply(urls, dot_every(10, delay_by(1, download.file)), quiet = TRUE)
```

Useful behavioural FOs

Implementing `delay_by` is straightforward, and follows the same basic template that we'll see for the majority of FOs in this chapter:

```
delay_by <- function(delay, f) {
  function(...) {
    Sys.sleep(delay)
    f(...)
  }
}
system.time(runif(100))
#>
#> 0.004 0.000 0.001
system.time(delay_by(1, runif)(100))
#>
#> 0.004 0.008 1.001
```

`dot_every` is a little bit more complicated because it needs to modify state in the parent environment using `<<-`. If it's not clear how this works, you might want to re-read the mutable state section in Functional programming[3].

```
dot_every <- function(n, f) {
  i <- 1
  function(...) {
    if (i %% n == 0) cat(".")
    i <<- i + 1
```

[3]Functional-programming.html

```
    f(...)
  }
}
x <- lapply(1:100, runif)
x <- lapply(1:100, dot_every(10, runif))
#> ..........
```

Notice that I've made the function the last argument to each FO. This make it reads a little better when we compose multiple function operators. If the function was the first argument, then instead of:

```
download <- dot_every(10, delay_by(1, download.file))
```

we'd have

```
download <- dot_every(delay_by(download.file, 1), 10)
```

which is a little harder to follow because the argument to `dot_every()` is far away from the its call. That's sometimes called the Dagwood sandwich[4] problem: you have too much filling (too many long arguments) between your slices of bread (parentheses). I've also tried to give my FOs names that you can read easily: delay by 1 (second), (print a) dot every 10 (invocations). The more clearly your code expresses your interent through function names, the easier it is for others (and future you) to understand the code.

Two other tasks that you can solve with a behaviour FO are:

- Logging a time stamp and message to a file everytime a function is run:

  ```
  log_to <- function(path, message, f) {
    stopifnot(file.exists(path))

    function(...) {
      cat(Sys.time(), ": ", message, sep = "", file = path,
        append = TRUE)
      f(...)
    }
  }
  ```

- Ensuring that if the first input is `NULL` then the output is `NULL` (the name is inspired by Haskell's maybe monad which fills a similar role in Haskell, making it possible for any function to work with a `NULL` argument).

[4]http://en.wikipedia.org/wiki/Dagwood_sandwich

```
maybe <- function(f) {
  function(x, ...) {
    if (is.null(x)) return(NULL)
    f(x, ...)
  }
}
```

Memoisation

Another thing you might worry about when downloading multiple files is accidentally downloading the same file multiple times. You could avoid it by calling `unique` on the list of input urls, or manually managing a data structure that mapped the url to the result. An alternative approach is to use memoisation: a way of modifying a function to automatically cache its results.

```
library(memoise)

slow_function <- function(x) {
  Sys.sleep(1)
  10
}
system.time(slow_function())
#>
#> 0.008 0.004 1.000
system.time(slow_function())
#>
#> 0.008 0.004 1.000
fast_function <- memoise(slow_function)
system.time(fast_function())
#>
#> 0.004 0.012 1.001
system.time(fast_function())
#>
#>     0     0     0
```

Memoisation is an example of a classic tradeoff in computer science: trading space for speed. A memoised function uses more memory (because it stores all of the previous inputs and outputs), but is much faster.

A somewhat more realistic use case is implementing the Fibonacci series (a topic we'll come back to [[software systems]]). The Fibonacci series is defined recursively: the first two values are 1 and 1, then f(n) = f(n - 1) + f(n - 2). A naive version implemented in R is very slow because (e.g.) `fib(10)` computes `fib(9)` and `fib(8)`, and `fib(9)` computes `fib(8)` and `fib(7)`, and so on, so that the value for each location gets computed many many times. Memoising `fib()`

makes the implementation much faster because each value is only computed once, and then remembered.

```
fib <- function(n) {
  if (n < 2) return(1)
  fib(n - 2) + fib(n - 1)
}
system.time(fib(23))
#>
#> 0.180 0.012 0.192
system.time(fib(24))
#>
#> 0.288 0.016 0.313

fib2 <- memoise(function(n) {
  if (n < 2) return(1)
  fib2(n - 2) + fib2(n - 1)
})
system.time(fib2(23))
#>
#> 0.004 0.000 0.006
system.time(fib2(24))
#>
#>     0     0     0
```

It doesn't make sense to memoise all functions. The example below shows that a memoised random number generator is no longer random:

```
runifm <- memoise(runif)
runifm(5)
#> [1] 0.9671 0.7691 0.8748 0.6358 0.5873
runifm(5)
#> [1] 0.9671 0.7691 0.8748 0.6358 0.5873
```

Once we understand `memoise()`, it's straightforward to apply it to our problem:

```
download <- dot_every(10, memoise(delay_by(1, download.file)))
```

This gives a function that we can easily use with `lapply()`. If something goes wrong with the loop inside `lapply()`, it can be difficult to tell what's going on; the next section shows how we can use FOs to open the curtain and look inside.

Capturing function invocations

One challenge with functionals is that it can be hard to see what's going on - it's not easy to pry open the internals like it is with a for loop. However, we can use FOs to help us. The `tee` function, defined below, has three arguments, all functions: `f`, the original function; `on_input`, a function that's called with the inputs to `f`, and `on_output` a function that's called with the output from `f`.

```
ignore <- function(...) NULL
tee <- function(f, on_input = ignore, on_output = ignore) {
  function(...) {
    input <- if (nargs() == 1) c(...) else list(...)
    on_input(input)
    output <- f(...)
    on_output(output)
    output
  }
}
```

(The function is inspired by the unix `tee` shell command which is used to split streams of file operations up so that you can see what's happening or save intermediate results to a file. It's named after the `t` connector in plumbing)

We can use `tee` to look into how `uniroot` finds where `x` and `cos(x)` intersect:

```
g <- function(x) cos(x) - x
zero <- uniroot(g, c(-5, 5))

# The location where the function is evaluated
zero <- uniroot(tee(g, on_input = print), c(-5, 5))
#> [1] -5
#> [1] 5
#> [1] 0.283662
#> [1] 0.875203
#> [1] 0.72298
#> [1] 0.738631
#> [1] 0.739085
#> [1] 0.739024
#> [1] 0.739085
# The value of the function
zero <- uniroot(tee(g, on_output = print), c(-5, 5))
#> [1] 5.28366
#> [1] -4.71634
#> [1] 0.676375
#> [1] -0.234363
#> [1] 0.0268568
#> [1] 0.00076012
#> [1] -0.000000260399
#> [1] 0.000101887
#> [1] -0.000000260399
```

Using `print()` allows us to see what's happening as the function runs, but it doesn't give us any ability to work with the values. Instead we might want to capture the sequence of the calls. To do that we create a function called `remember()` that remembers every argument it was called with, and retrieves them when coerced into a list. (The small amount of S3 magic that makes this simple is explained in the [S3] chapter).

```
remember <- function() {
  memory <- list()
  f <- function(...) {
    # This is inefficient!
    memory <<- append(memory, list(...))
    invisible()
  }

  structure(f, class = "remember")
}
as.list.remember <- function(x, ...) {
  environment(x)$memory
}
print.remember <- function(x, ...) {
  cat("Remembering...\n")
  str(as.list(x))
}
```

Now we can see exactly how uniroot zeros in on the final answer:

```
locs <- remember()
vals <- remember()
zero <- uniroot(tee(g, locs, vals), c(-5, 5))
# FIXME: should need as.list.remember, but knitr environment
# seems to prevent S3 from finding the right method
x <- sapply(as.list.remember(locs), "[[", 1)
error <- sapply(as.list.remember(vals), "[[", 1)
plot(x, type = "b"); abline(h = 0.739, col = "grey50")

plot(error, type = "b"); abline(h = 0, col = "grey50")
```

Exercises

- What does the following function do? What would be a good name for it?

  ```
  f <- function(g) {
    result <- NULL
    function(...) {
      if (is.null(result)) {
        result <<- g(...)
      }
      result
    }
  }
  runif2 <- f(runif)
  runif2(5)
  #> [1] 0.4088 0.2884 0.5393 0.5695 0.4350
  runif2(5)
  #> [1] 0.4088 0.2884 0.5393 0.5695 0.4350
  ```

- Modify `delay_by()` so that instead of delaying by a fixed amount of time, it ensures that a certain amount of time has elapsed since the function was last called. That is, if you called `g <- delay_by(1, f); g(); Sys.sleep(2); g()` there shouldn't be an extra delay.

- Write `wait_until()` which delays execution until a specific time. Or write `run_after()` which only runs a function after a specified time, returning `NULL` otherwise.

- There are three places we could have added a memoise call: why did we choose the one we did?

  ```
  download <- memoise(dot_every(10, delay_by(1, download.file)))
  download <- dot_every(10, memoise(delay_by(1, download.file)))
  download <- dot_every(10, delay_by(1, memoise(download.file)))
  ```

- Why is the `remember()` function inefficient? How could you implement it in more efficient way?

Output FOs

The next step up in complexity is to modify the output of a function. This could be quite simple, or it could fundamentally change the operation of the function, returning something completely different to its usual output. In this section you'll learn about two simple modifications, `Negate()` and `failwith()`, and two fundamental modifications, `capture_it()` and `time_it()`.

Minor modifications

`base::Negate` and `plyr::failwith` offer two minor, but useful, modifications of a function that are particularly handy in conjunction with functionals.

`Negate` takes a function that returns a logical vector (a predicate function), and returns the negation of that function. This can be a useful shortcut when the function you have returns the opposite of what you need. Its essence is very simple:

```
Negate <- function(f) {
  function(...) !f(...)
}
(Negate(is.null))(NULL)
#> [1] FALSE
```

I often use this idea to make a `compact()` function that removes all null elements from a list:

```
compact <- function(x) Filter(Negate(is.null), x)
```

`plyr::failwith()` turns a function that throws an error into a function that returns a default value when there's an error. Again, the essence of `failwith()` is simple, it's just a wrapper around `try()`, which captures errors and continues execution. (if you haven't seen `try()` before, it's discussed in more detail in the exceptions and debugging[5] chapter):

```
failwith <- function(default = NULL, f, quiet = FALSE) {
  function(...) {
    out <- default
    try(out <- f(...), silent = quiet)
    out
  }
}
log("a")
#> Error:
failwith(NA, log)("a")
#> [1] NA
failwith(NA, log, quiet = TRUE)("a")
#> [1] NA
```

`failwith()` is very useful in conjunction with functionals: instead of the failure propagating and terminating the higher-level loop, you can complete the iteration and then find out what went wrong. For example, imagine you're fitting a

[5]Exceptions-Debugging.html

set of generalised linear models (glms) to a list of data frames. Sometimes glms fail because of optimisation problems. You still want to try to fit all the models, then once that's complete, look at the data sets that failed to fit:

```
# If any model fails, all models fail to fit:
models <- lapply(datasets, glm, formula = y ~ x1 + x2 * x3)
# If a model fails, it will get a NULL value
models <- lapply(datasets, failwith(NULL, glm),
  formula = y ~ x1 + x2 * x3)

# remove failed models (NULLs) with compact
ok_models <- compact(models)
# use where to extract the datasets corresponding to failed models
failed_data <- datasets[where(models, is.null)]
```

I think this is a great example of the power of combining functionals and function operators: it makes it easy to succinctly express what you need to solve a common data analysis problem.

Changing what a function does

Other output function operators can have a more profound affect on the operation of the function. Instead of returning the original return value, we can return some other effect of the function evaluation. Here's two examples:

- Return text that the function `print()`ed:

  ```
  capture_it <- function(f) {
    function(...) {
      capture.output(f(...))
    }
  }
  str_out <- capture_it(str)
  str(1:10)
  #>  int [1:10] 1 2 3 4 5 6 7 8 9 10
  str_out(1:10)
  #> [1] " int [1:10] 1 2 3 4 5 6 7 8 9 10"
  ```

- Return how long a function took to run:

  ```
  time_it <- function(f) {
    function(...) {
      system.time(f(...))
    }
  }
  ```

`time_it()` allows us to rewrite some of the code from the functionals chapter:

```
compute_mean <- list(
  base = function(x) mean(x),
  sum = function(x) sum(x) / length(x)
)
x <- runif(1e6)

# Instead of using an anonymous function to time
lapply(compute_mean, function(f) system.time(f(x)))
#> $base
#>
#> 0.004 0.000 0.005
#>
#> $sum
#>
#> 0.000 0.000 0.002

# We can compose function operators
call_fun <- function(f, ...) f(...)
lapply(compute_mean, time_it(call_fun), x)
#> $base
#>
#> 0.004 0.000 0.005
#>
#> $sum
#>
#> 0.000 0.000 0.003
```

In this example, there's not a huge benefit to using function operators, because the composition is simple and we're applying the same operator to each function. Generally, using function operators is more effective when you are using multiple operators or if the gap between creating them and using them is large.

Exercises

- Create a `negative` function that flips the sign of the output from the function to which it's applied.

- The `evaluate` package makes it easy to capture all the outputs (results, text, messages, warnings, errors and plots) from an expression. Create a function like `capture_it()` that also captures the warnings and errors generated by a function.

- Create a FO that tracks files created or deleted in the working directory (Hint: use `setDiff()` and `dir()`). What other global effects do functions have you might want to track?
- Modify the final example to use `fapply()` from the functionals[6] chapter instead of `lapply()`.

Input FOs

The next step up in complexity is to modify the inputs of a function.Again, you can modify how a function works in a minor way (e.g., prefilling some of the arguments), or fundamentally change the inputs (e.g. converting inputs from scalar to vector, or vector to matrix).

Prefilling function arguments: partial function application

A common use of anonymous functions is to make a variant of a function that has certain arguments "filled in" already. This is called "partial function application", and is implemented by `pryr::partial`. (Once you have read the computing on the language chapter, I encourage you to read the source code for `partial` and puzzle out how it works - it's only 5 lines of code!)

`partial()` allows us to replace code like

```
f <- function(a) g(a, b = 1)
compact <- function(x) Filter(Negate(is.null), x)
Map(function(x, y) f(x, y, zs), xs, ys)
```

with

```
f <- partial(g, b = 1)
compact <- partial(Filter, Negate(is.null))
Map(partial(f, zs = zs), xs, ys)
```

We can use this idea to simplify some of the code we used when working with lists of functions. Instead of:

```
funs2 <- list(
  sum = function(x, ...) sum(x, ..., na.rm = TRUE),
  mean = function(x, ...) mean(x, ..., na.rm = TRUE),
  median = function(x, ...) median(x, ..., na.rm = TRUE)
)
```

[6] Functionals.html

We can write:

```
library(pryr)
funs2 <- list(
  sum = partial(sum, na.rm = TRUE),
  mean = partial(mean, na.rm = TRUE),
  median = partial(median, na.rm = TRUE)
)
```

But if you look closely you'll notice we're just applying a function to every element in a list, and that's the job of `lapply`. This allows us to reduce the code still further::

```
funs <- c(sum = sum, mean = mean, median = median)
funs2 <- lapply(funs, partial, na.rm = TRUE)
```

Let's think about a similar, but subtly different case. Say we have a numeric vector and we want to generate a list of trimmed means with that amount of trimming. The following code doesn't work because we want the first argument of `partial` to be fixed to mean. We could try specifying the argument name because fixed matching overrides positional, but that doesn't work because the `trims` end up supplied to the first argument of `mean`.

```
(trims <- seq(0, 0.9, length = 5))
#> [1] 0.000 0.225 0.450 0.675 0.900
funs3 <- lapply(trims, partial, `_f` = mean)
sapply(funs3, call_fun, c(1:100, (1:50) * 100))
#> Error: 'trim'
```

Instead we could use an anonymous function

```
funs4 <- lapply(trims, function(t) partial(mean, trim = t))
funs4[[1]]
#> function (...)
#> mean(trim = t, ...)
#> <environment: 0x14386148>
sapply(funs4, call_fun, c(1:100, (1:50) * 100))
#> [1] 75.5 75.5 75.5 75.5 75.5
```

But that doesn't work because each function gets a promise to evaluate `t`, and that promise isn't evaluated until all of the functions are run, by which time `t = 0.9`. To make it work you need to manually force the evaluation of t:

```
funs5 <- lapply(trims, function(t) {
  force(t)
  partial(mean, trim = t)
})
funs5[[1]]
#> function (...)
#> mean(trim = t, ...)
#> <environment: 0x1409ee68>
sapply(funs5, call_fun, c(1:100, (1:50) * 100))
#> [1] 883.7 235.6  75.5  75.5  75.5
```

When writing functionals, you can expect your users to know of `partial()` and not use `...` For example, instead of implementing `lapply()` like:

```
lapply2 <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
unlist(lapply2(1:5, log, base = 10))
#> [1] 0.0000 0.3010 0.4771 0.6021 0.6990
```

we could implement it as:

```
lapply3 <- function(x, f) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]])
  }
  out
}
unlist(lapply3(1:5, partial(log, base = 10)))
#> [1] 0.0000 0.3010 0.4771 0.6021 0.6990
```

Partial function application is straightforward in many functional programming languages, but it's not entirely clear how it should interact with R's lazy evaluation rules. The approach of `plyr::partial` takes is to create a function as similar as possible to the anonymous function you'd create by hand. Peter Meilstrup takes a different approach in his ptools package[7]; you might want to read about `%()%`, `%>>%` and `%<<%` if you're interested in the topic.

[7] https://github.com/crowding/ptools/

Changing input types

Instead of a minor change to the function's inputs, it's also possible to make a function work with a fundamentally different type of data. There are a few existing functions along these lines:

- `base::Vectorize` converts a scalar function to a vector function. `Vectorize` takes a non-vectorised function and vectorises with respect to the arguments given in the `vectorizge.args` parameter. This doesn't give you any magical performance improvements, but it is useful if you want a quick and dirty way of making a vectorised function.

  A mildly useful extension of `sample` would be to vectorize it with respect to size: this would allow you to generate multiple samples in one call.

  ```
  sample2 <- Vectorize(sample, "size", SIMPLIFY = FALSE)
  sample2(1:5, c(1, 1, 3))
  #> [[1]]
  #> [1] 1
  #>
  #> [[2]]
  #> [1] 1
  #>
  #> [[3]]
  #> [1] 5 2 1
  sample2(1:5, 5:3)
  #> [[1]]
  #> [1] 3 4 5 2 1
  #>
  #> [[2]]
  #> [1] 1 4 3 5
  #>
  #> [[3]]
  #> [1] 4 3 2
  ```

  In this example we have used `SIMPLIFY = FALSE` to ensure that our newly vectorised function always returns a list. This is usually what you want.

- `splat` converts a function that takes multiple arguments to a function that takes a single list of arguments.

  ```
  splat <- function (f) {
    function(args) {
      do.call(f, args)
    }
  }
  ```

  This is useful if you want to invoke a function with varying arguments:

```
x <- c(NA, runif(100), 1000)
args <- list(
  list(x),
  list(x, na.rm = TRUE),
  list(x, na.rm = TRUE, trim = 0.1)
)
lapply(args, splat(mean))
#> [[1]]
#> [1] NA
#>
#> [[2]]
#> [1] 10.38
#>
#> [[3]]
#> [1] 0.4897
```

- `plyr::colwise()` converts a vector function to one that works with data frames:

```
median(mtcars)
#> Error: need numeric data
median(mtcars$mpg)
#> [1] 19.2
plyr::colwise(median)(mtcars)
#>    mpg cyl  disp  hp  drat    wt  qsec vs am gear carb
#> 1 19.2   6 196.3 123 3.695 3.325 17.71  0  0    4    2
```

Exercises

- Our previous `download()` function will only download a single file. How can you use `partial()` and `lapply()` to create a function that downloads multiple files at once? What are the pros and cons of using `partial()` vs. writing a function by hand?

- Read the source code for `plyr::colwise()`. How does code work? It performs three main tasks. What are they? How could you make `colwise` simpler by implementing each separate task as a function operator? (Hint: think about `partial`)

- Write FOs that convert a function to return a matrix instead of a data frame, or a data frame instead of a matrix. (If you already know [S3], make these methods of `as.data.frame` and `as.matrix`)

- You've seen five functions that modify a function to change it's output from one form to another. What are they? Draw a table: what should go in the rows and what should go in the columns? What function operators might you want to write to fill in the missing cells? Come up with example use cases.

- Look at all the examples of using an anonymous function to partially apply a function in this and the previous chapter. Replace the anonymous function with `partial`. What do you think of the result? Is it easier or harder to read?

Combining FOs

Instead of operating on single functions, function operators can take multiple functions as input. One simple example of this is `plyr::each()` which takes a list of vectorised functions and returns a single function that applies each in turn to the input:

```
summaries <- plyr::each(mean, sd, median)
summaries(1:10)
#>   mean     sd median
#>  5.500  3.028  5.500
```

Two more complicated examples are combining functions through composition, or through boolean algebra. These are glue that join multiple functions together.

Function composition

An important way of combining functions is through composition: `f(g(x))`. Composition takes a list of functions and applies them sequentially to the input. It's a replacement for the common anonymous function pattern where you chain together multiple functions to get the result you want:

```
sapply(mtcars, function(x) length(unique(x)))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

A simple version of compose looks like this:

```
compose <- function(f, g) {
  function(...) f(g(...))
}
```

(`pryr::compose()` provides a fuller-featured alternative that can accept multiple functions).

This allows us to write:

```
sapply(mtcars, compose(length, unique))
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6
```

Mathematically, function composition is often denoted with an infix operator, o, `(f o g)(x)`. Haskell, a popular functional programming language, uses `.` in a similar manner. In R, we can create our own infix function that works similarly:

```
"%.%" <- compose
sapply(mtcars, length %.% unique)
#>  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
#>   25    3   27   22   22   29   30    2    2    3    6

sqrt(1 + 8)
#> [1] 3
compose(sqrt, `+`)(1, 8)
#> [1] 3
(sqrt %.% `+`)(1, 8)
#> [1] 3
```

Compose also allows for a very succinct implement of `Negate`: it's just a partially evaluated version of `compose()`.

```
Negate <- partial(compose, `!`)
```

We could also implement the standard deviation by breaking it down into a separate set of function compositions:

```
square <- function(x) x ^ 2
deviation <- function(x) x - mean(x)

sd <- sqrt %.% mean %.% square %.% deviation
sd(1:10)
#> [1] 2.872
```

This type of programming is called tacit or point-free programming. (The term point free comes from use the of the word point to refer values in topology; this style is also derogatorily known as pointless). In this style of programming you don't explicitly refer to variables, focussing on the high-level composition of functions, rather than the low-level flow of data. Since we're using only functions and not parameters, we use verbs and not nouns, and this style leads to code that focusses on what's being done, not what it's being done to. This style is common in Haskell, and is the typical style in stack based programming languages like Forth and Factor. It's not a terribly natural or elegant style in R, but it is a useful tool to have in your toolbox.

`compose()` is particularly useful in conjunction with `partial()`, because `partial()` allows you to supply additional arguments to the functions being composed. One nice side effect of this style of programming is that it keeps the arguments to each function near the function name. This is important because code gets harder to understand as the size of the chunk of code you have to hold in your head grows.

Below I take the example from the first section of the chapter and modify it to use the two styles of function composition defined above. They are both longer than the original code but maybe easier to understand because the function and its arguments are closer together. Note that we still have to read them from right to left (bottom to top): the first function called is the last one written. We could define `compose()` to work in the opposite direction, but in the long run, this is likely to lead to confusion since we'd create a small part of the langugage that reads differently to every other part.

```
download <- dot_every(10, memoise(delay_by(1, download.file)))

download <- pryr::compose(
  partial(dot_every, 10),
  memoise,
  partial(delay_by, 1),
  download.file
)

download <- partial(dot_every, 10) %.%
  memoise %.%
  partial(delay_by, 1) %.%
  download.file
```

Logical predicates and boolean algebra

When I use `Filter()` and other functionals that work with logical predicates, I often find myself using anonymous functions to combine multiple conditions:

```
Filter(function(x) is.character(x) || is.factor(x), iris)
```

As an alternative, we could define some function operators that combine logical predicates:

```
and <- function(f1, f2) {
  function(...) {
    f1(...) && f2(...)
  }
}
or <- function(f1, f2) {
  function(...) {
    f1(...) || f2(...)
  }
}
not <- function(f1) {
  function(...) {
    !f1(...)
  }
}
```

which would allow us to write:

```
Filter(or(is.character, is.factor), iris)
```

This allows us to express arbitrarily complicated boolean expressing involving functions in a succinct way.

Exercises

- Implement your own version of `compose` using `Reduce` and `%.%`. For bonus points, do it without calling `function`.
- Extend `and()` and `or()` to deal with any number of input functions. Can you do it with `Reduce()`? Can you keep them lazy (so e.g. for `and()` the function returns as soon as it sees the first `FALSE`)?
- Implement the `xor()` binary operator. Implement it using the existing `xor()` function. Implement it as a combination of `and()` and `or()`. What are the advantages and disadvantages of each approach? Also think about what you'll call the resulting function, and how you might need to change the names of `and()`, `not()` and `or()` in order to keep them consistent.
- Above, we implemented boolean algebra for functions that return a logical function. Implement elementary algebra (`plus()`, `minus()`, `multiply()`, `divide()`, `exponentiate()`, `log()`) for functions that return numeric vectors.

The common pattern and a subtle bug

Most function operators we've seen follow a similar pattern:

```
funop <- function(f, otherargs) {
  function(...) {
    # maybe do something
    res <- f(...)
    # maybe do something else
    res
  }
}
```

There's a subtle problem with this implementation. It does not work well with `lapply()` because `f` is lazily evaluated. This means that if you give `lapply()` a list of functions and a FO to apply it to each of them, it will look like it repeatedly applied the FO to the last function:

```
wrap <- function(f) {
  function(...) f(...)
}
fs <- list(sum = sum, mean = mean, min = min)
gs <- lapply(fs, wrap)
gs$sum(1:10)
#> [1] 1
environment(gs$sum)$f
#> function (..., na.rm = FALSE)  .Primitive("min")
```

Another problem is that as designed, we have to pass in a funtion object, not the name of a function, which is often convenient. We can solve both problems by using `match.fun()`: it forces evaluation of `f`, and will find the function object if given its name:

```
wrap2 <- function(f) {
  f <- match.fun(f)
  function(...) f(...)
}
fs <- c(sum = "sum", mean = "mean", min = "min")
hs <- lapply(fs, wrap2)
hs$sum(1:10)
#> [1] 55
environment(hs$sum)$f
#> function (..., na.rm = FALSE)  .Primitive("sum")
```

Exercises

- Why does the following code (from stackoverflow[8]) not do what you expect?

[8]http://stackoverflow.com/questions/8440675

```
a <- list(0, 1)
b <- list(0, 1)

# return a linear function with slope a and intercept b.
f <- function(a, b) function(x) a * x + b

# create a list of functions with different parameters.
fs <- Map(f, a, b)

fs[[1]](3)
#> [1] 4
```

How can you modify `f` so that it works correctly?

Part III

Computing on the language

Chapter 12

Metaprogramming

‘’Flexibility in syntax, if it does not lead to ambiguity, would seem a reasonable thing to ask of an interactive programming language.” — Kent Pitman, http://www.nhplace.com/kent/Papers/Special-Forms.html

R has powerful tools for computing not only on values, but also on the actions that lead to those values. These tools are powerful and magical, and one of the most surprising features if you’re coming from another programming language. Take the following simple snippet of code that draws a sine curve:

```
x <- seq(0, 2 * pi, length = 100)
sinx <- sin(x)
plot(x, sinx, type = "l")
```

Look at the labels on the axes! How did R know that the variable on the x axis was called `x` and the variable on the y axis was called `sinx`? In most programming languages, you can only access values of the arguments provided to functions, but in R you can also access the expression used to compute them. Combined with R’s lazy evaluation mechanism this gives function authors considerable power to both access the underlying expression and do special things with it.

Techniques based on these tools are generally called “computing on the language”, and in R provide a set of tools with power equivalent to functional and object oriented programming. This chapter will introduce you to the basic ideas of special evaluation, show you how they are used in base R, and how you can use them to create your own functions that save typing for interactive analysis. These tools are very useful for developing convenient user-facing functions because they can dramatically reduce the amount of typing required to specify an action.

Computing on the language is an extremely powerful tool, but it can also create code that is hard for others to understand and is substantially harder to program with. Before you use it, make sure that you have exhausted all other possibilities. You'll also learn about the downsides: because these tools work with expressions rather than values this increases ambiguity in the function call, and makes the function difficult to call from another function.

The following chapters [[expressions]] and [[special-environments]], expand on these ideas, discussing the underlying data structures and how you can understand and manipulate them to create new tools.

In this chapter you'll learn:

- how many functions such as `plot()` and `data.frame()` capture the names of the variable supplied to them, and the downsides of this technique.
- Manipulate a data frame by referring to the variables: `subset()`, `transform()`, `plyr::mutate()`, `plyr::arrange()`, `plyr::summarise()`, `with()`
- Work around non-standard evaluation (like lattice functions) with `substitute`.
- Capture an expression for later evaluation: ggplot2 and plyr
- Use formulas to describe computations: `lm()` and the lattice package

Capturing expressions

The tool that makes non-standard evaluation possible in R is `substitute()`. It looks at a function argument, and instead of seeing the value, it looks to see how the value was computed:

```
f <- function(x) {
  substitute(x)
}
f(1:10)
#> 1:10
f(x)
#> x
f(x + y ^ 2 / z + exp(a * sin(b)))
#> x + y^2/z + exp(a * sin(b))
```

We won't worry yet about exactly what sort of object `substitute()` returns (that's the topic of the [Expressions] chapter), but we'll call it an expression.

(Note that it's not the same thing as returned by the `expression()` function: we'll call that an expression *object*.)

`substitute()` works because function arguments in R are only evaluated when they are needed, not automatically when the function is called. This means that function arguments are not just a simple value, but instead store both the expression to compute the value and the environment in which to compute it. Together these two things are called a **promise**. Most of the time in R, you don't need to know anything about promises because the first time you access a promise it is seamlessly evaluated, returning its value.

We need one more function if we want to understand how `plot()` and `data.frame()` work: `deparse()`. This function takes an expression and converts it to a character vector.

```
g <- function(x) deparse(substitute(x))
g(1:10)
#> [1] "1:10"
g(x)
#> [1] "x"
g(x + y ^ 2 / z + exp(a * sin(b)))
#> [1] "x + y^2/z + exp(a * sin(b))"
```

There's one important caveat with `deparse()`: it can return a multiple strings if the input is long:

```
g(a + b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q + r + s + t + u + v
```

If you need a single string, you can work around this by using the `width.cutoff` argument (which has a maximum value of 500), or by joining the lines back together again with `paste()`.

You might wonder why we couldn't use our original `f()` to compute `g()`. Let's try it:

```
g <- function(x) deparse(f(x))
g(1:10)
#> [1] "x"
g(x)
#> [1] "x"
g(x + y ^ 2 / z + exp(a * sin(b)))
#> [1] "x"
```

This is one of the downsides of functions that use `substitute()`: because they use the expression, not the value, of an argument, it becomes harder to call them from other functions. We'll talk more about this problem and some remedies later on.

There are a lot of functions in base R that use these ideas. Some use them to avoid quotes:

```
library(ggplot2)
library("ggplot2")
```

Others use them to provide default labels. For example, `plot.default()` has code that basically does (the real code is more complicated because of the way base plotting methods work, but it's effectively the same):

```
plot.default <- function(x, y = NULL, xlabel = NULL, ylabel = NULL, ...) {
    ...
    xlab <- if (is.null(xlabel) && !missing(x)) deparse(substitute(x))
    ylab <- if (is.null(xlabel) && !missing(y)) deparse(substitute(y))
    ...
}
```

If a label is not set and the variable is present, then the expression used to generate the value of `x` is used as a default value for the label on the x axis.

`data.frame()` does a similar thing. It automatically labels variables with the expression used to compute them:

```
x <- 1:4
y <- letters[1:4]
names(data.frame(x, y))
#> [1] "x" "y"
```

This wouldn't be possible in most programming langauges because functions usually only see values (e.g. `1:4` and `c("a", "b", "c", "d")`), not the expressions that created them (`x` and `y`).

Non-standard evaluation in subset

Just printing out the expression used to generate an argument value is useful, but we can do even more with the unevaluated function. For example, take `subset()`. It's a useful interactive shortcut for subsetting data frames: instead of repeating the data frame you're working with again and again, you can save some typing:

```
subset(mtcars, cyl == 4)
# equivalent to:
# mtcars[mtcars$cyl == 4, ]

subset(mtcars, vs == am)
# equivalent to:
# mtcars[mtcars$vs == mtcars$am, ]
```

Subset is special because `vs == am` or `cyl == 4` aren't evaluated in the global environment: instead they're evaluated in the data frame. In other words, `subset()` implements different [[scoping|Scoping]] rules so instead of looking for those variables in the current environment, `subset()` looks in the specified data frame. This is called **non-standard evaluation**: you are deliberately breaking R's usual rules in order to do something special.

How does `subset()` work? We've already seen how to capture the expression that represents an argument, rather than its value, so we just need to figure out how to evaluate that expression in the right context: i.e. `cyl` should be interpreted as `mtcars$cyl`. To do this we need `eval()`, which takes an expression and evaluates it in the specified environment.

But before we can do that, we need to learn one more useful function: `quote()`. It's similar to `substitute()` but it always gives you back exactly the expression you entered. This makes it useful for interactive experimentation.

```
quote(1:10)
#> 1:10
quote(x)
#> x
quote(x + y ^ 2 / z + exp(a * sin(b)))
#> x + y^2/z + exp(a * sin(b))
```

Now let's experiment with `eval()`. If you only provide one argument, it evaluates the expression in the current environment. This makes `eval(quote(x))` exactly equivalent to typing `x`, regardless of what `x` is:

```
eval(quote(x <- 1))
eval(quote(x))
#> [1] 1

eval(quote(cyl))
#> Error:    'cyl'
```

Note that `quote()` and `eval()` are basically opposites. In the example below, each `eval()` peels off one layer of quoting.

```
quote(2 + 2)
#> 2 + 2
eval(quote(2 + 2))
#> [1] 4

quote(quote(2 + 2))
#> quote(2 + 2)
eval(quote(quote(2 + 2)))
#> 2 + 2
eval(eval(quote(quote(2 + 2))))
#> [1] 4
```

What will this code return?

```
eval(quote(eval(quote(eval(quote(2 + 2))))))
```

The second argument to `eval()` controls which environment the code is evaluated in:

```
x <- 10
eval(quote(x))
#> [1] 10

e <- new.env()
e$x <- 20
eval(quote(x), e)
#> [1] 20
```

Instead of an environment, the second argument can also be a list or a data frame. This works because an environment is basically a set of mappings between names and values, in the same way as a list or data frame.

```
eval(quote(x), list(x = 30))
#> [1] 30
eval(quote(x), data.frame(x = 40))
#> [1] 40
```

This is basically what we want for `subset()`:

```
eval(quote(cyl == 4), mtcars)
#>  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
#> [12] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
#> [23] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE
eval(quote(vs == am), mtcars)
```

```
#>  [1] FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
#> [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
#> [23]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

We can combine `eval()` and `substitute()` together to write `subset()`: we can capture the call representing the condition, evaluate it in the context of the data frame, and then use the result for subsetting:

```
subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x)
  x[r, ]
}
subset2(mtcars, cyl == 6)
#>                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
#> Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
#> Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
#> Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
#> Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
```

When you first start using `eval()` it's easy to make mistakes. Here's a common one: forgetting to quote the input:

```
eval(cyl, mtcars)
#> Error:    'cyl'
# Carefully look at the difference to this error
eval(quote(cyl), mtcars)
#>  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

Exercises

- The real subset function (`subset.data.frame()`) does two other things to the result. What are they?
- The other component of the real subset function is variable selection. It allows you to work with variable names like they are positions, so you can do things like `subset(mtcars, , -cyl)` to drop the cylinder variable, or `subset(mtcars, , disp:drat)` to select all the variables between `disp` and `drat`. How does select work? I've made it easier to understand by extracting it out into its own function:

```
select <- function(df, vars) {
  vars <- substitute(vars)
  var_pos <- setNames(as.list(seq_along(df)), names(df))
  pos <- eval(vars, var_pos)
  df[, pos, drop = FALSE]
}
select(mtcars, -cyl)
#>                      mpg  disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4           21.0 160.0 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag       21.0 160.0 110 3.90 2.875 17.02  0  1    4    4
#> Datsun 710          22.8 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Hornet 4 Drive      21.4 258.0 110 3.08 3.215 19.44  1  0    3    1
#> Hornet Sportabout   18.7 360.0 175 3.15 3.440 17.02  0  0    3    2
#> Valiant             18.1 225.0 105 2.76 3.460 20.22  1  0    3    1
#> Duster 360          14.3 360.0 245 3.21 3.570 15.84  0  0    3    4
#> Merc 240D           24.4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230            22.8 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Merc 280            19.2 167.6 123 3.92 3.440 18.30  1  0    4    4
#> Merc 280C           17.8 167.6 123 3.92 3.440 18.90  1  0    4    4
#> Merc 450SE          16.4 275.8 180 3.07 4.070 17.40  0  0    3    3
#> Merc 450SL          17.3 275.8 180 3.07 3.730 17.60  0  0    3    3
#> Merc 450SLC         15.2 275.8 180 3.07 3.780 18.00  0  0    3    3
#> Cadillac Fleetwood  10.4 472.0 205 2.93 5.250 17.98  0  0    3    4
#> Lincoln Continental 10.4 460.0 215 3.00 5.424 17.82  0  0    3    4
#> Chrysler Imperial   14.7 440.0 230 3.23 5.345 17.42  0  0    3    4
#> Fiat 128            32.4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Honda Civic         30.4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> Toyota Corolla      33.9  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Toyota Corona       21.5 120.1  97 3.70 2.465 20.01  1  0    3    1
#> Dodge Challenger    15.5 318.0 150 2.76 3.520 16.87  0  0    3    2
#> AMC Javelin         15.2 304.0 150 3.15 3.435 17.30  0  0    3    2
#> Camaro Z28          13.3 350.0 245 3.73 3.840 15.41  0  0    3    4
#> Pontiac Firebird    19.2 400.0 175 3.08 3.845 17.05  0  0    3    2
#> Fiat X1-9           27.3  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Porsche 914-2       26.0 120.3  91 4.43 2.140 16.70  0  1    5    2
#> Lotus Europa        30.4  95.1 113 3.77 1.513 16.90  1  1    5    2
#> Ford Pantera L      15.8 351.0 264 4.22 3.170 14.50  0  1    5    4
#> Ferrari Dino        19.7 145.0 175 3.62 2.770 15.50  0  1    5    6
#> Maserati Bora       15.0 301.0 335 3.54 3.570 14.60  0  1    5    8
#> Volvo 142E          21.4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

- What does `evalq()` do? Use it to reduce the amount of typing for the examples above that use both `eval()` and `quote()`

Scoping issues

While it certainly looks like our `subset2()` function works, whenever we're working with expressions instead of values, we need to test a little more carefully. For example, you might expect that the following uses of `subset2()` should all return the same value because each variable refers to the same value:

```
y <- 4
x <- 4
condition <- 4
condition_call <- 4

subset2(mtcars, cyl == 4)
#>                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
#> Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
#> Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#> Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
subset2(mtcars, cyl == y)
#>                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
#> Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
#> Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#> Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
subset2(mtcars, cyl == x)
#>       mpg cyl disp hp drat wt qsec vs am gear carb
#> NA     NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.1   NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.2   NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.3   NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.4   NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.5   NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.6   NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.7   NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.8   NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.9   NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.10  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.11  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.12  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.13  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.14  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.15  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.16  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.17  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.18  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.19  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.20  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.21  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.22  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.23  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.24  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.25  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.26  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.27  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.28  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.29  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.30  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.31  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.32  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.33  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.34  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.35  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.36  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.37  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.38  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.39  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.40  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
#> NA.41  NA  NA   NA NA   NA NA   NA NA NA   NA   NA
subset2(mtcars, cyl == condition)
#> Error:    'cyl'
subset2(mtcars, cyl == condition_call)
#> Warning:

#>  [1] mpg  cyl  disp hp   drat wt   qsec vs   am   gear carb
#> <0  > ( 0-  row.names)
```

What's going wrong? You can get a hint from the variable names I've chosen: they are all variables defined inside `subset2()`. It seems like if `eval()` can't find the variable inside the data frame (it's second argument), it's looking in the function environment. That's obviously not what we want, so we need some way to tell `eval()` to look somewhere else if it can't find the variables in the data frame.

The key is the third argument: `enclos`. This allows us to specify the parent (or enclosing) environment for objects that don't have one like lists and data frames (`enclos` is ignored if we pass in a real environment). The `enclosing` environment is where any objects that aren't found in the data frame will be looked for. By default it uses the environment of the current function, which is not what we want.

We want to look for `x` in the environment in which `subset` was called. In R terminology this is called the **parent frame** and is accessed with `parent.frame()`. This is an example of dynamic scope[1]. With this modification our function works:

```
subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x, parent.frame())
  x[r, ]
}

x <- 4
subset2(mtcars, cyl == x)
#>                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
#> Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
#> Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#> Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Using `enclos` is just a short cut for converting a list or data frame to an environment with the desired parent yourself. We can use the `list2env()` to turn a list into an environment and explicitly set the parent ourselves:

```
subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
```

[1] http://en.wikipedia.org/wiki/Scope_%28programming%29#Dynamic_scoping

```
  env <- list2env(x, parent = parent.frame())
  r <- eval(condition_call, env)
  x[r, ]
}

x <- 4
subset2(mtcars, cyl == x)
#>                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Toyota Corona  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
#> Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
#> Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#> Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

When evaluating code in a non-standard way, it's also a good idea to test your code works when run outside of the global environment:

```
f <- function() {
  x <- 6
  subset(mtcars, cyl == x)
}
f()
#>                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
#> Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
#> Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
#> Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
#> Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
```

And indeed it now works.

Exercises

- `plyr::arrange()` works similarly to `subset()`, but instead of selecting rows, it reorders them. How does it work? What does `substitute(order(...))` do?

- What does `transform()` do? (Hint: read the documentation). How does it work? (Hint: read the source code for `transform.data.frame`) What does `substitute(list(...))` do? (Hint: create a function that does only that and experiment with it).
- `plyr::mutate()` is similar to `transform()` but it applies the transformations sequentially so that transformation can refer to columns that were just created:

  ```
  df <- data.frame(x = 1:5)
  transform(df, x2 = x * x, x3 = x2 * x)
  plyr::mutate(df, x2 = x * x, x3 = x2 * x)
  ```

How does mutate work? What's the key difference between mutate and transform?

- What does `with()` do? How does it work? (Read the source code for `with.default()`)
- What does `within()` do? How does it work? (Read the source code for `within.data.frame()`). What makes the code so much more complicated than `with()`?

Calling from another function

Typically, computing on the language is most useful for functions called directly by the user, not by other functions. While `subset` saves typing, it has one big disadvantage: it's now difficult to use non-interactively, e.g. from another function. For example, you might try using `subset()` from within a function that is given the name of a variable and it's desired value:

```
colname <- "cyl"
val <- 6

subset(mtcars, colname == val)
#>  [1] mpg  cyl  disp hp   drat wt   qsec vs   am   gear carb
#> <0 > ( 0-  row.names)
# Zero rows because "cyl" != 6
```

Or imagine we want to create a function that randomly reorders a subset of the data. A nice way to write that function would be to write a function for random reordering and a function for subsetting (that we already have!) and combine the two together. Let's try that:

```
scramble <- function(x) x[sample(nrow(x)), ]

subscramble <- function(x, condition) {
  scramble(subset(x, condition))
}
```

But when we run that we get:

```
subscramble(mtcars, cyl == 4)
#> Error:     'cyl'
# Error in eval(expr, envir, enclos) : object 'cyl' not found
traceback()
#> 22: vapply(x, f, logical(1)) at <text>#2
#> 21: where(iris, is.factor) at <text>#2
#> 20: eval(expr, envir, enclos)
#> 19: eval(call, envir, enclos)
#> 18: withVisible(eval(call, envir, enclos))
#> 17: withCallingHandlers(withVisible(eval(call, envir, enclos)), warning = wHandler,
#>         error = eHandler, message = mHandler)
#> 16: handle(ev <- withCallingHandlers(withVisible(eval(call, envir,
#>         enclos)), warning = wHandler, error = eHandler, message = mHandler))
#> 15: evaluate_call(expr, parsed$src[[i]], envir = envir, enclos = enclos,
#>         debug = debug, last = i == length(out), use_try = stop_on_error !=
#>             2L, keep_warning = keep_warning, keep_message = keep_message,
#>         output_handler = output_handler)
#> 14: evaluate(code, envir = env, new_device = FALSE, keep_warning = !isFALSE(options$w
#>         keep_message = !isFALSE(options$message), stop_on_error = if (options$error &
#>             options$include) 0L else 2L)
#> 13: in_dir(opts_knit$get("root.dir") %n% input_dir(), evaluate(code,
#>         envir = env, new_device = FALSE, keep_warning = !isFALSE(options$warning),
#>         keep_message = !isFALSE(options$message), stop_on_error = if (options$error &
#>             options$include) 0L else 2L))
#> 12: block_exec(params)
#> 11: call_block(x)
#> 10: process_group.block(group)
#> 9: process_group(group)
#> 8: withCallingHandlers(if (tangle) process_tangle(group) else process_group(group),
#>        error = function(e) {
#>            cat(res, sep = "\n", file = output %n% "")
#>            message("Quitting from lines ", paste(current_lines(i),
#>                collapse = "-"), " (", knit_concord$get("infile"),
#>                ") ")
#>        })
#> 7: process_file(text, output)
#> 6: knit(in_path) at rmd2html.r#52
#> 5: rmd2md(chapter) at build-book.r#9
#> 4: eval(expr, envir, enclos)
#> 3: eval(ei, envir)
#> 2: withVisible(eval(ei, envir))
#> 1: source("./book/build-book.r")
# 5: eval(expr, envir, enclos)
# 4: eval(condition_call, x)
# 3: subset(x, condition)
# 2: scramble(subset(x, condition))
# 1: subscramble(mtcars, cyl == 4)
```

What's gone wrong? To figure it out, lets `debug()` subset and work through the code line-by-line:

```
> debugonce(subset)
> subscramble(mtcars, cyl == 4)
debugging in: subset(x, condition)
debug: {
    condition_call <- substitute(condition)
    r <- eval(condition_call, x)
    x[r, ]
}
Browse[2]> n
debug: condition_call <- substitute(condition)
Browse[2]> n
debug: r <- eval(condition_call, x)
Browse[2]> condition_call
condition
Browse[2]> eval(condition_call, x)
Error in eval(expr, envir, enclos) : object 'cyl' not found
Browse[2]> condition
Error: object 'cyl' not found
In addition: Warning messages:
1: restarting interrupted promise evaluation
2: restarting interrupted promise evaluation
```

Can you see what the problem is? `condition_call` contains the expression `condition` so when we try to evaluate that it evaluates `condition` which has the value `cyl == 4`. This can't be computed in the parent environment because it doesn't contain an object called `cyl`. If `cyl` is set in the global environment, far more confusing things can happen:

```
cyl <- 4
subscramble(mtcars, cyl == 4)
#>                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
#> Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
#> Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
#> Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
#> Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
#> Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
#> Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
#> Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
#> Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
#> Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
#> AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
#> Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
#> Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
#> Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
#> Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
#> Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
#> Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
#> Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
#> Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
#> Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
#> Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
#> Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
#> Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
#> Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#> Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
```

```
cyl <- sample(10, 100, rep = T)
subscramble(mtcars, cyl == 4)
#>               mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> NA.6           NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
#> NA.3           NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
#> NA.2           NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
#> NA.4           NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
#> NA.5           NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
#> NA.7           NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
#> NA             NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
#> Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#> NA.1           NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
#> Volvo 142E   21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
#> NA.9            NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
#> Camaro Z28    13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
#> NA.8            NA  NA    NA  NA   NA    NA    NA NA NA   NA   NA
```

This is an example of the general tension between functions that are designed for interactive use, and functions that are safe to program with. A function that uses `substitute()` might save typing, but it's difficult to call from another function. As a developer you should also provide an alternative version that works when passed a quoted expression. For example, we could rewrite:

```
subset2_q <- function(x, condition) {
  r <- eval(condition, x, parent.frame())
  x[r, ]
}

subset2 <- function(x, condition) {
  subset2_q(x, substitute(condition))
}

subscramble <- function(x, condition) {
  condition <- substitute(condition)
  scramble(subset2_q(x, condition))
}
```

I usually suffix these functions with `q` to indicate that they take a quoted call. Most users won't need them so the name can be a little longer.

You might wonder why the function couldn't do this automatically:

```
subset <- function(x, condition) {
  if (!is.call(condition)) {
    condition <- substitute(condition)
  }
  r <- eval(condition, x)
  x[r, ]
}
subset(mtcars, quote(cyl == 4))
subset(mtcars, cyl == 4)
```

But hopefully a little thought, or maybe some experimentation, will show why this doesn't work.

Substitute

Following the examples above, whenever you write your own functions that use non-standard evaluation, you should always provide alternatives that others can use. But what happens if you want to call a function that uses non-standard evaluation and doesn't have a form that takes expressions? For example, imagine you want to create a lattice graphic given the names of two variables:

```
library(lattice)
xyplot(mpg ~ disp, data = mtcars)


x <- quote(mpg)
y <- quote(disp)
xyplot(x ~ y, data = mtcars)
#> Error:   'symbol'
```

Again, we can turn to substitute and use it for another purpose: modifying expressions. So far we've just used `substitute()` to capture the unevaluated expression associated with arguments, but it can actually do much much more, and is a very useful for manipulating expressions in general.

Unfortunately `substitute()` has a "feature" that makes experimenting with it interactively a bit of a pain: it never does substitutions when run from the global environment, and just behaves like `quote()`:

```
a <- 1
b <- 2
substitute(a + b + x)
#> 1 + 2 + mpg
```

But if we run it inside a function, `substitute()` substitutes what it can and leaves everything else the same:

```
f <- function() {
  a <- 1
  b <- 2
  substitute(a + b + x)
}
f()
#> 1 + 2 + x
```

To make it easier to experiment with `substitute()`, `pryr` provides the `subs()` function. It works exactly the same way as `substitute()` except it has a shorter name and if the second argument is the global environment it turns it into a list. Together, this makes it much easier to experiment with substitution:

```
library(pryr)
subs(a + b + x)
#> 1 + 2 + mpg
```

The second argument (to both `sub()` and `substitute()`) can override the use of the current environment, and provide an alternative list of name-value pairs to use. The following example uses that technique to show some variations on substituting a string, variable name or function call:

```
subs(a + b, list(a = "y"))
#> "y" + b
subs(a + b, list(a = quote(y)))
#> y + b
subs(a + b, list(a = quote(y())))
#> y() + b
```

Remember that every action in R is a function call, so we can also replace `+` with another function:

```
subs(a + b, list("+" = quote(f)))
#> f(a, b)
subs(a + b, list("+" = quote(`*`)))
#> a * b
```

Note that it's quite possible to make nonsense commands with `substitute`:

```
subs(y <- y + 1, list(y = 1))
#> 1 <- 1 + 1
```

And you can use substitute to insert any arbitrary object into an expression. This is technically ok, but often results in surprisingly and undesirable behaviour. In the example below, the expression we create doesn't print correctly, but it returns the correct result when we evaluate it:

```
df <- data.frame(x = 1)
(x <- subs(class(df)))
#> class(list(x = 1))
eval(x)
#> [1] "data.frame"
```

Formally, substitution takes place by examining each name in the expression. If the name refers to:

- an ordinary variable, it's replaced by the value of the variable.
- a promise, it's replaced by the expression associated with the promise.
- `...`, it's replaced by the contents of `...` (only if the substitution occurs in a function)

Otherwise the name is left as is.

We can use this to create the right call to `xyplot`:

```r
x <- quote(mpg)
y <- quote(disp)
subs(xyplot(x ~ y, data = mtcars))
#> xyplot(mpg ~ disp, data = mtcars)
```

It's even simpler inside a function, because we don't need to explicitly quote the x and y variables. Following the rules above, `substitute()` replaces named arguments with their expressions, not their values:

```r
xyplot2 <- function(x, y, data = data) {
  substitute(xyplot(x ~ y, data = data))
}
xyplot2(mpg, disp, data = mtcars)
#> xyplot(mpg ~ disp, data = mtcars)
```

If we include `...` in the call to substitute, we can add additional arguments to the call:

```r
xyplot3 <- function(x, y, ...) {
  substitute(xyplot(x ~ y, ...))
}
xyplot3(mpg, disp, data = mtcars, col = "red", aspect = "xy")
#> xyplot(mpg ~ disp, data = mtcars, col = "red", aspect = "xy")
```

Non-standard evaluation in substitute

One application of this idea is to make a version of `substitute` that evaluates its first argument (i.e. a version that uses standard evaluation). Note the following example:

```r
x <- quote(a + b)
substitute(x, list(a = 1, b = 2))
#> x
```

Instead we can use `pryr::substitute2`:

```r
x <- quote(a + b)
substitute2(x, list(a = 1, b = 2))
#> 1 + 2
```

The implementation of `substitute2` is short, but deep:

```
substitute2 <- function(x, env) {
  call <- substitute(substitute(y, env), list(y = x))
  eval(call)
}
```

Let's work through the example above: `substitute2(x, list(a = 1, b = 2))`. It's a little tricky because of `substitute()`'s non-standard evaluation rules, we can't use the usual technique of working through the parentheses inside-out.

1. First `substitute(substitute(y, env), list(y = x))` is evaluated. The first argument is specially evaluated in the environment containing only one item, the value of `x` with the name `y`. Because we've put `x` inside a list, it will be evaluated and the rules of substitute will replace `y` with it's value. This yields the expression `substitute(a + b, env)`
2. Next we evaluate that expression inside the current function. `substitute()` specially evaluates its first argument, and looks for name value pairs in `env`, which evaluates to `list(a = 1, b = 2)`. Those are both values (not promises) so the result will be `a + b`

Capturing unevaluated ...

Another frequently useful technique is to capture all of the unevaluated expressions in `...`. Base R functions do this in many ways, but there's one technique that works well in a wide variety of situations:

```
dots <- function(...) {
  eval(substitute(alist(...)))
}
```

This uses the `alist()` function which simply captures all its arguments. This function is the same as `pryr::dots()`, and pryr also provides `pryr::named_dots()`, which ensures all arguments are named, using the deparsed expressions as default names.

The downsides of non-standard evaluation

There are usually two principles you can follow when modelling the evaluation of R code:

- If the underlying values are the same, the results will the same. i.e. the three results will all be the same:

  ```
  x <- 10; y <- 10
  f(10); f(x); f(y)
  ```

- You can model evaluation by working from the innermost parentheses to the outermost.

Non-standard evaluation can break both principles. This makes the mental model needed to correctly predict the output much more complicated, so it's only worthwhile to do so if there is significant gain.

For example, `library()` and `require()` allow you to call them either with or without quotes, because internally they use `deparse(substitute(x))` plus a couple of tricks. That means that these two lines do exactly the same thing:

```
library(ggplot2)
library("ggplot2")
```

However, things start to get complicated if the variable has a value.. What do you think the following lines of code will do?

```
ggplot2 <- "plyr"
library(ggplot2)
```

It loads ggplot2, not plyr. If you want to load plyr (the value of the ggplot2 variable), you need to use an additional argument:

```
library(ggplot2, character.only = TRUE)
```

Using an argument to change the behaviour of another argument is not a great idea because it means you must completely and carefully read all of the function arguments to understand what one function argument means. You can't understand the effect of each argument in isolation, and hence it's harder to reason about the results of a function call.

There are a number of other R functions that use `substitute()` and `deparse()` in this way: `ls()`, `rm()`, `data()`, `demo()`, `example()`, `vignette()`. These all use non-standard evaluation and then have a special ways of enforcing the usual rules. To me, eliminating two quotes is not worth the cognitive cost of non-standard evaluation, and I don't recommend you use `substitute()` for this purpose.

One situtation where non-standard evaluation is more useful is `data.frame()`, which uses the input expressions to automatically name the output variables if not otherwise provided:

```
x <- 10
y <- "a"
df <- data.frame(x, y)
names(df)
#> [1] "x" "y"
```

I think it is worthwhile in `data.frame()` because it eliminates a lot of redundancy in the common scenario when you're creating a data frame from existing variables, and importantly, it's easy to override this behaviour by supplying names for each variable.

The code for `data.frame()` is rather complicated, but we can create our own simple version for lists to see how a function that does this might work. The key is `pryr::named_dots()`, a function which returns the unevaluated ... arguments, with default names. Then it's just a matter of arranging the evaluated results in a list:

```
list2 <- function(...) {
  dots <- named_dots(...)
  lapply(dots, eval, parent.frame())
}
x <- 1; y <- 2
list2(x, y)
#> $x
#> [1] 1
#>
#> $y
#> [1] 2
list2(x, z = y)
#> $x
#> [1] 1
#>
#> $z
#> [1] 2
```

Applications

To show how I've used some of these ideas in practice, the following two sections show applications of non-standard evaluation to plyr and ggplot2.

`plyr::.` and `ggplot2::aes`

Both plyr and ggplot2 have ways of capturing what you want to do, and then performing that action later. ggplot2 uses the `aes()` to define a set of mappings between variables in your data and visual properties on your graphic. plyr uses the `.` function to capture the names (or more complicated expressions) of variables used to split a data frame into pieces. Let's look at the code:

```
. <- function (..., .env = parent.frame()) {
  structure(
    as.list(match.call()[-1]),
    env = .env,
    class = "quoted"
  )
}

aes <- function (x = NULL, y = NULL, ...) {
  aes <- structure(
    as.list(match.call()[-1]),
    class = "uneval")
  class(aes) <- "uneval"
  ggplot2:::rename_aes(aes)
}
```

Both functions were written when I didn't know so much about non-standard evaluation, and if I was to write them today, I'd use the `dots()` helper function I showed previously. I'd also think more about the environment in which the results of `aes()` should be evaluated, and how that integrates with ggplot2's rules for aesthetic mapping inheritance. That's a bit murky at the moment and leads to confusion when creating complex graphics across multiple functions.

ggplot2 and plyr provide slightly different ways to use standard evaluation so that you can refer to variables by reference. ggplot2 provides `aes_string()` which allows you to specify variables by the string representation of their name, and plyr uses S3 methods so that you can either supply an object of class quoted (as created with `.()`), or a regular character vector.

Plyr: summarise, mutate and arrange

The plyr package also uses non-standard evaluation to complete the set of tools provided by the base `subset()` and `transform()` functions with `mutate()`, `summarise()` and `arrange()`. Each of these functions has the same interface: the first argument is a data frame and the subsequent arguments are evaluated in the context of that data frame (i.e. they look there first for variables, and then in the current environment) and they return a data frame.

The following code shows the essence of how these four functions work:

```
subset2 <- function(.data, subset) {
  sub <- eval(substitute(subset), .data, parent.frame())
  sub <- sub & !is.na(sub)

  .data[sub, , drop = FALSE]
}
arrange2 <- function (.data, ...) {
  ord <- eval(substitute(order(...)), .data, parent.frame())
  .data[ord, , drop = FALSE]
}

mutate2 <- function(.data, ...) {
  cols <- named_dots(...)
  data_env <- eval_df(.data, parent.frame(), cols)

  out_cols <- union(names(.data), names(cols))
  quickdf(mget(out_cols, data_env))
}
summarise2 <- function (.data, ...) {
  cols <- named_dots(...)
  data_env <- eval_df(.data, parent.frame(), cols)

  quickdf(mget(names(cols), env))
}
eval_df <- function(data, env, expr) {
  data_env <- list2env(data, parent = env)

  for(nm in names(exprs)) {
    data_env[[nm]] <- eval(data_env[[nm]], env)
  }
  data_env
}
```

You might be surprised to see the for loops in `eval_df`, but they are necessary because the computation of one variable might depend on the results of previous variables (this is the key difference between `mutate()` and `transform()`).

Combined with a by operator (e.g. `ddply()`) these four functions allow you to express the majority of data manipulation operations. Then when you have a new problem, solving it becomes a matter of thinking about which operations you need to apply and in what order. The realm of possible actions has been shrunk to a manageable number.

Conclusion

Now that you understand how our version of subset works, go back and read the source code for `subset.data.frame`, the base R version which does a little more.

Other functions that work similarly are `with.default`, `within.data.frame`, `transform.data.frame`, and in the plyr package `.`, `arrange`, and `summarise`. Look at the source code for these functions and see if you can figure out how they work.

Chapter 13

Expressions

In [[computing on the language]], you learned the basics of accessing the expressions underlying computation in R, and evaluating them in new ways. In this chapter, you'll learn more about the underlying structure of expressions, and how you can compute on them directly.

- The structure of expressions (a tree made up of constants, names and calls) and how you can create and modify them directly

- How to flexibly convert expressions between their tree form and their text form, and how `source()` works.

- Create functions by hand as an alternative instead of using a closure, so that viewing the source of the function shows something meaningful

- Walk the code tree using recursive functions to understand how many of the functions in the codetools package work, and to you write your own functions that detect if a function uses logical abbreviations, list all assignments inside a function and understand how `bquote()` works.

It's generally a bad idea to create code by operating on its string representation: there is no guarantee that you'll create valid code. Don't get me wrong: pasting strings together will often allow you to solve your problem in the least amount of time, but it may create subtle bugs that will take your users hours to track down. Learning more about the structure of the R language and the tools that allow you to modify it is an investment that will pay off by allowing you to make more robust code.

Thoroughout this chapter we're going to use tools from the `pryr` package to help see what's going on. If you don't already have it, install it by running `devtools::install_github("pryr")`

Structure of expressions

To compute on the language, we first need to be understand the structure of the language. That's going to require some new vocabulary, some new tools and some new ways of thinking about R code. The first thing you need to understand is the distinction between an operation and its result:

```
x <- 4
y <- x * 10
y
#> [1] 40
```

We want to distinguish between the action of multiplying x by 10 and assigning the results to `y` versus the actual result (40). In R, we can capture the action with `quote()`:

```
z <- quote(y <- x * 10)
z
#> y <- x * 10
```

`quote()` gives us back an **expression**, an object that represents an action that can be performed by R. (Confusingly the `expression()` function produces expression lists, but since you'll never need to use that function we can safely ignore it).

An expression is also called an abstract syntax tree (AST) because it represents the abstract structure of the code in a tree form. We can use `pryr::call_tree()` to see the hierarchy more clearly:

```
library(pryr)
call_tree(z)
#> \- <-()
#>   \- `y
#>   \- *()
#>     \- `x
#>     \- 10
```

There are three basic things you'll commonly see in a call tree: constants, names and calls.

- **constants** are atomic vectors, like `"a"` or `10`. These appear as is.

  ```
  call_tree(quote("a"))
  #> \- "a"
  ```

```
call_tree(quote(1))
#> \- 1
call_tree(quote(1L))
#> \- 1L
call_tree(quote(TRUE))
#> \- TRUE
```

- **names** which represent the name of a variable, not its value. (Names are also sometimes called symbols). These are prefixed with '

```
call_tree(quote(x))
#> \- `x
call_tree(quote(mean))
#> \- `mean
call_tree(quote(`an unusual name`))
#> \- `an unusual name
```

- **calls** represent the action of calling a function, not its result. These are suffixed with (). The arguments to the function are listed below it, and can be constants, names or other calls.

```
call_tree(quote(f()))
#> \- f()
call_tree(quote(f(1, 2)))
#> \- f()
#>   \- 1
#>   \- 2
call_tree(quote(f(a, b)))
#> \- f()
#>   \- `a
#>   \- `b
call_tree(quote(f(g(), h(1, a))))
#> \- f()
#>   \- g()
#>   \- h()
#>     \- 1
#>     \- `a
```

As mentioned in Functions[1], even things that don't look like function calls still follow this same hierarchical structure:

```
call_tree(quote(a + b))
#> \- +()
#>   \- `a
#>   \- `b
```

[1] Functions.html

```
call_tree(quote(if (x > 1) x else 1/x))
#> \- if()
#>   \- >()
#>     \- `x
#>     \- 1
#>   \- `x
#>   \- /()
#>     \- 1
#>     \- `x
call_tree(quote(function(x, y) {x * y}))
#> \- function()
#>   \- list(x = , y = )
#>   \- {()
#>     \- *()
#>       \- `x
#>       \- `y
#>   \- structure(c(3L, 17L, 3L, 38L, 17L, 38L, 3L, 3L), srcfile = <environ...
```

(In general, it's possible for any type of R object to live in a call tree, but these are the only three types you'll get from parsing R code. It's possible to put anything else inside an expression using the tools described below, but while technically correct, support is often patchy.)

Together, names and calls are sometimes called language objects, and can be tested for with `is.language()`. Note that `str()` is somewhat inconsistent with respect to this naming convention, describing names as symbols, and calls as a language objects:

```
str(quote(a))
#>  symbol a
str(quote(a + b))
#>  language a + b
```

Together, constants, names and calls define the structure of all R code. The following section provides more detail about each in turn.

Constants

Quoting a single atomic vector gives it back to you:

```
is.atomic(quote(1))
#> [1] TRUE
identical(1, quote(1))
#> [1] TRUE
```

```r
is.atomic(quote("test"))
#> [1] TRUE
identical("test", quote("test"))
#> [1] TRUE
```

But quoting a vector of values gives you something different because you always use a function to create a vector:

```r
identical(1:3, quote(1:3))
#> [1] FALSE
identical(c(FALSE, TRUE), quote(c(FALSE, TRUE)))
#> [1] FALSE
```

It's possible to use `substitute()` to directly insert a vector into a call tree, but use this with caution as you are creating a call that is not be generated during the normal operation of R.

```r
y <- substitute(f(x), list(x = 1:3))
is.atomic(y)
#> [1] FALSE
```

Names

As well as capturing names with `quote()`, it's also possible to convert strings to names. This is mostly useful when your function receives strings as input, as it's more typing than using `quote()`:

```r
as.name("name")
#> name
identical(quote(name), as.name("name"))
#> [1] TRUE

as.name("a b")
#> `a b`
```

Note that the second example produces the name `` `a b `‘: the backticks are the standard way of escape non-standard names in R. (And are explained more in Functions[2])

There's one special name that needs a little extra discussion: the name that represents missing values. You can get this from the formals of a function, or with `alist()`:

[2]Functions.html

```
formals(plot)$x

alist(x =)[[1]]
```

It is basically a special name, that you can create by using `quote()` in a slightly unusual way:

```
quote(expr =)
```

Note that this object is behaves strangely, and is rarely useful, except when (as we'll see later) you want to try a function with arguments that don't have defaults.

```
x <- quote(expr =)
x
#> Error:    "x",
```

Calls

As well as capturing complete calls using `quote()`, modifying existing calls using `substitute()` you can also create calls from their constituent pieces using using `as.call()` or `call()`.

The first argument to `call()` is a string giving a function name, and the other arguments should be expressions that are used as the arguments to the call.

```
call(":", 1, 10)
#> 1:10
call("mean", 1:10, na.rm = TRUE)
#> mean(1:10, na.rm = TRUE)
```

`as.call()` is a minor variation that takes a list where the first argument is the *name* of a function (not a string), and the subsequent values are the arguments.

```
x_call <- quote(1:10)
mean_call <- as.call(list(quote(mean), x_call))
identical(mean_call, quote(mean(1:10)))
#> [1] TRUE
```

Note that the following two calls look the same, but are actually different:

```
(a <- call("mean", 1:10))
#> mean(1:10)
(b <- call("mean", quote(1:10)))
#> mean(1:10)
identical(a, b)
#> [1] FALSE
call_tree(a)
#> \- mean()
#>   \- 1:10
call_tree(b)
#> \- mean()
#>   \- :()
#>     \- 1
#>     \- 10
```

In a, the first argument to mean is an integer vector containing the numbers 1 to 10, and in b the first argument is a call to :. You can put any R object into an expression, but the printing of expression objects will not always show the difference.

The key difference is where/when evaluation occurs:

```
a <- call("print", Sys.time())
b <- call("print", quote(Sys.time()))
eval(a); Sys.sleep(1); eval(a)
#> [1] "2013-12-22 20:23:12 CST"
#> [1] "2013-12-22 20:23:12 CST"
eval(b); Sys.sleep(1); eval(b)
#> [1] "2013-12-22 20:23:13 CST"
#> [1] "2013-12-22 20:23:14 CST"
```

The first element of a call doesn't have to be the name of a function, and instead can be a call that generates a function:

```
(function(x) x + 1)(10)
#> [1] 11
add <- function(y) function(x) x + y
add(1)(10)
#> [1] 11

# Note that you can't create this sort of call with call
call("add(1)", 10)
#> `add(1)`(10)
# But you can with as.call
as.call(list(quote(add(1)), 10))
#> add(1)(10)
```

An interesting use of `call()`

One interesting use of call lies inside the `mode<-` function which is an alternative way to change the mode of a vector. The important part of the function is extracted in the `mode2<-` function below.

```
`mode2<-` <- function (x, value) {
  mde <- paste0("as.", value)
  eval(call(mde, x), parent.frame())
}
x <- 1:10
mode2(x) <- "character"
x
#>  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Another way to achieve the same goal would be find the function and then call it:

```
`mode3<-` <- function(x, value) {
  mde <- match.fun(paste0("as.", value))
  mde(x)
}
x <- 1:10
mode3(x) <- "character"
x
#>  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Generally, I'd prefer `mode3<-` over `mode2<-` because it uses concepts familiar to more R programmers, and generally it's a good idea to use the simplest and most commonly understood techniques that solve a given problem.

Extracting elements of a call

When it comes to modifying calls, they behave almost exactly like lists: a call has `length`, `'[[` and `[` methods. The length of a call minus 1 gives the number of arguments:

```
x <- quote(read.csv("important.csv", row.names = FALSE))
length(x) - 1
#> [1] 2
```

The first element of the call is the *name* of the function:

```
x[[1]]
#> read.csv
# read.csv
```

The remaining elements are the arguments to the function, which can be extracted by name or by position.

```
x$row.names
#> [1] FALSE
x[[3]]
#> [1] FALSE

names(x)
#> [1] ""          ""          "row.names"
```

Standardising function calls

Generally, extracting arguments by position is dangerous, because R's function calling semantics are so flexible. It's better to match by name, but all arguments might not be named. The solution to this problem is to use `match.call()`, which takes a function and a call as arguments:

```
y <- match.call(read.csv, x)
names(y)
#> [1] ""          "file"      "row.names"

# Or if you don't know in advance what the function is
match.call(eval(x[[1]]), x)
#> read.csv(file = "important.csv", row.names = FALSE)
```

This will be an important tool when we start manipulating existing function calls. If we don't use `match.call` we'll need a lot of extra code to deal with all the possible ways to call a function.

We can wrap this up into a function. To figure out the definition of the associated function we evaluate the first component of the call, the name of the function. We need to specify an environment here, because the function might be different in different places. Whenever we provide an environment parameter, `parent.frame()` is usually a good default.

Note the check for primitive functions: they don't have `formals()` and handle argument matching specially, so there's nothing we can do.

```
standardise_call <- function(call, env = parent.frame()) {
  stopifnot(is.call(call))
  f <- eval(call[[1]], env)
  if (is.primitive(f)) return(call)

  match.call(f, call)
}

standardise_call(y)
#> read.csv(file = "important.csv", row.names = FALSE)
standardise_call(quote(standardise_call(y)))
#> standardise_call(call = y)
```

Modifying a call

You can add, modify and delete elements of the call with the standard replacement operators, `$<-` and `[[<-`:

```
y$row.names <- TRUE
y$col.names <- FALSE
y
#> read.csv(file = "important.csv", row.names = TRUE, col.names = FALSE)

y[[2]] <- "less-important.csv"
y[[4]] <- NULL
y
#> read.csv(file = "less-important.csv", row.names = TRUE)

y$file <- quote(paste0(filename, ".csv"))
y
#> read.csv(file = paste0(filename, ".csv"), row.names = TRUE)
```

Calls also support the `[` method, but use it with care: since the first element is the function to call, removing it is unlikely to create a call that will evaluate without error.

```
x[-3] # remove the second argument
#> read.csv("important.csv")
x[-1] # remove the function name - but it's still a call!
#> "important.csv"(row.names = FALSE)
x
#> read.csv("important.csv", row.names = FALSE)
```

If you want to get a list of the unevaluated arguments, explicitly convert it to a list:

```
# A list of the unevaluated arguments
as.list(x[-1])
#> [[1]]
#> [1] "important.csv"
#>
#> $row.names
#> [1] FALSE
```

We can use these ideas to create an easy way modify a call given a list.

```
modify_call <- function(call, new_args) {
  call <- standardise_call(call)
  nms <- names(new_args) %||% rep("", length(new_args))

  if (any(nms == "")) {
    stop("All new arguments must be named", call. = FALSE)
  }

  for(nm in nms) {
    call[[nm]] <- new_args[[nm]]
  }
  call
}
modify_call(quote(mean(x, na.rm = TRUE)), list(na.rm = NULL))
#> mean(x = x)
modify_call(quote(mean(x, na.rm = TRUE)), list(na.rm = FALSE))
#> mean(x = x, na.rm = FALSE)
modify_call(quote(mean(x, na.rm = TRUE)), list(x = quote(y)))
#> mean(x = y, na.rm = TRUE)
```

Exercises

- If you create a functional, you may want it to accept the name of a function as a string or the name of a function. Use `substitute()` and what you know about expressions to create a function that returns a list containing the name of the function (where you can determine it) and the function itself.

  ```
  fname(mean)
  list(name = "mean", f = mean)
  fname("mean")
  list(name = "mean", f = mean)
  fname(function(x) sum(x) / length(x))
  list(name = "<anonymous>", f = function(x) sum(x) / length(x))
  ```

Create a version that uses standard evaluation suitable for calling from another function (Hint: it should have two arguments: an expression and an environment).

Parsing and deparsing

You can convert quoted calls back and forth between text with `parse()` and `deparse()`. You've seen `deparse()` already it: takes an expression and returns a character vector. `parse()` does the opposite: it takes a character vector and returns a list of expressions, also known as an expression object or expression list.

Note that because the primary use of `parse()` is parsing files of code on disk, the first argument is a file path, and if you have the code in a character vector, you need to use the `text` argument.

```
z <- quote(y <- x * 10)
deparse(z)
#> [1] "y <- x * 10"

parse(text = deparse(z))
#> expression(y <- x * 10)
```

`deparse()` returns a character vector with an entry for each line, and by default it will try to make lines that are around 60 characters long. If you want a single string be sure to `paste()` it back together, and read the other options in the documentation.

`parse()` can't return just a single expression, because there might be many top-level calls in an file. So instead it returns expression objects, or expression lists. You should never need to create expression objects yourself, and all you need to know about them is that they're a list of calls:

```
exp <- parse(text = c("x <- 4\ny <- x * 10"))
length(exp)
#> [1] 2
exp[[1]]
#> x <- 4
is.call(exp[[1]])
#> [1] TRUE
call_tree(exp)
#> \- <-()
#>   \- `x
#>   \- 4
```

```
#>
#> \- <-()
#>   \- `y
#>   \- *()
#>     \- `x
#>     \- 10
```

It's not possible for `parse()` and `deparse()` be completely symmetric. See the help for `deparse()` for more details.

Sourcing files from disk

With `parse()` and `eval()` you can write your own simple version of `source()`. We read in the file on disk, `parse()` it and then `eval()` each component in the specified environment. This version defaults to a new environment, so it doesn't affect existing objects. `source()` invisibly returns the result of the last expression in the file, so `simple_source()` does the same.

```
simple_source <- function(file, envir = new.env()) {
  stopifnot(file.exists(file))
  stopifnot(is.environment(envir))

  lines <- readLines(file, warn = FALSE)
  exprs <- parse(text = lines, n = -1)

  n <- length(exprs)
  if (n == 0L) return(invisible())

  for (i in seq_len(n - 1)) {
    eval(exprs[i], envir)
  }
  invisible(eval(exprs[n], envir))
}
```

The real `source()` is considerably more complicated because it preserves the underlying source code, can `echo` input and output, and has many additional settings to control behaviour.

Exercises

Capturing the current call

You may want to capture the expression that caused the current function to be run. There are two ways to do this:

- `sys.call()` captures exactly what the user typed
- `match.call()` uses R's regular argument matching rules and converts everything to full name matching. This is usually easier to work with because you know that the call will always have the same structure.

The following example illustrates the difference:

```
f <- function(abc = 1, def = 2, ghi = 3, ...) {
  list(sys = sys.call(), match = match.call())
}
f(d = 2, 2)
#> $sys
#> f(d = 2, 2)
#>
#> $match
#> f(abc = 2, def = 2)
```

A cautionary tale: `write.csv`

`write.csv()` is a base R function where call manipulation is used in a suboptimal manner. It captures the call to `write.csv()` and mangles it to instead call `write.table()`:

```
write.csv <- function (...) {
  Call <- match.call(expand.dots = TRUE)
  for (argname in c("append", "col.names", "sep", "dec", "qmethod")) {
    if (!is.null(Call[[argname]])) {
      warning(gettextf("attempt to set '%s' ignored", argname), domain = NA)
    }
  }
  rn <- eval.parent(Call$row.names)
  Call$append <- NULL
  Call$col.names <- if (is.logical(rn) && !rn) TRUE else NA
  Call$sep <- ","
  Call$dec <- "."
  Call$qmethod <- "double"
  Call[[1L]] <- as.name("write.table")
  eval.parent(Call)
}
```

We could write a function that behaves identically using regular function call semantics:

```
write.csv <- function(x, file = "", sep = ",", qmethod = "double", ...) {
  write.table(x = x, file = file, sep = sep, qmethod = qmethod, ...)
}
```

This makes the function much easier to understand: it's just calling `write.table` with different defaults. This also fixes a subtle bug in the original `write.csv`: `write.csv(mtcars, row = FALSE)` raises an error, but `write.csv(mtcars, row.names = FALSE)` does not. There's also no reason that `write.csv` shouldn't accept the `append` argument. Generally, you always want to use the simplest tool that will solve a problem - that makes it more likely that others will understand your code. Again, there's no point in using non-standard evaluation unless there's a big win: non-standard evaluation will make your function behave much less predictably.

Other uses of call capturing

Many modelling functions use `match.call()` to capture the call used to create the model. (This is one reason that creating lists of models using a function doesn't give the greatest output). This makes it possible to `update()` a model, modifying only a few components of the original model (but note that it doesn't preserve any of the computation, even if possible). Here's a quick example of `update()` in case you haven't used it before:

```
mod <- lm(mpg ~ wt, data = mtcars)
update(mod, formula = . ~ . + cyl)
#>
#> Call:
#> lm(formula = mpg ~ wt + cyl, data = mtcars)
#>
#> Coefficients:
#> (Intercept)           wt          cyl
#>       39.69        -3.19        -1.51
update(mod, subset = cyl == 4)
#>
#> Call:
#> lm(formula = mpg ~ wt, data = mtcars, subset = cyl == 4)
#>
#> Coefficients:
#> (Intercept)           wt
#>       39.57        -5.65
```

How does `update()` work? We can rewrite it using some of the tools (`dots()` and `modify_call()`) we've developed in this chapter to make it easier to see exactly what's going on. Once you've figured out what's going on here, you might want to read the source code for `update.default()` and see if you can how each component corresponds between the two versions.

```r
update_call <- function (object, formula., ...) {
  call <- object$call

  # Use a update.formula to deal with formulas like . ~ .
  if (!missing(formula.)) {
    call$formula <- update.formula(formula(object), formula.)
  }

  modify_call(call, dots(...))
}
update2 <- function(object, formula., ...) {
  call <- update_call(object, formula., ...)
  eval(call, parent.frame())
}
update_call(mod, formula = . ~ . + cyl)
#> lm(formula = mpg ~ wt + cyl, data = mtcars)
update_call(mod, subset = cyl == 4)
#> lm(formula = mpg ~ wt, data = mtcars, subset = cyl == 4)
```

The original `update()` has an `evaluate` argument that controls whether the function returns a call or the result, but I think it's good principle for a function to only return one type of object (not different types depending on the arguments) so I split it into two.

This rewrite also allows us to fix a small bug in update: it evaluates the call in the global environment, when really we want to re-evaluate it in the environment where the model was originally fit. This happens to be stored in the formula (called terms) so we can easily extract it.

```r
f <- function() {
  n <- 3
  lm(mpg ~ poly(wt, n), data = mtcars)
}
mod <- f()
update(mod, data = mtcars)
#> Error:    (3)

update2 <- function(object, formula., ...) {
  call <- update_call(object, formula., ...)
  eval(call, environment(object$terms))
}
update2(mod, data = mtcars)
#>
```

```
#> Call:
#> lm(formula = mpg ~ poly(wt, n), data = mtcars)
#>
#> Coefficients:
#>  (Intercept)  poly(wt, n)1  poly(wt, n)2  poly(wt, n)3
#>       20.091       -29.116         8.636         0.275
```

This is a good principle to remember: if you want to later replay the code you've captured using `match.call()` you really also need to capture the environment in which the code was evaluated.

There is a big potential downside: because you've captured that environment and saved it in an object, that environment will hang around and any objects in the environment will also hang around. That can have big implications for memory use. For example, in the following code, the big `x` and `y` objects will be captured in memory.

```
f <- function() {
  x <- runif(1e7)
  y <- runif(1e7)

  lm(mpg ~ wt, data = mtcars)
}
mod <- f()
object.size(environment(mod$terms)$x)
#> 80000040 bytes
```

Exercises

- Create a version of lm that doesn't do any special evaluation: all arguments should be quoted expressions, and it should construct a call to `lm` that preserves all information.

Creating a function

There's one function call that's so special it's worth devoting a little extra attention to: the `function` function that creates functions. This is one place we'll see pairlists (the object type that predated lists in R's history). The arguments of a function are stored as a pairlist: for our purposes we can treat a pairlist like a list, but we need to remember to cast arguments with `as.pairlist()`.

```
str(quote(function(x, y = 1) x + y)[[2]])
#> Dotted pair list of 2
#>  $ x: symbol
#>  $ y: num 1
```

Building up a function by hand is also useful when you can't use a closure because you don't know in advance what the arguments will be. We'll use `pryr::make_function` to build up a function from its component pieces: an argument list, a quoted body (the code to run) and the environment in which it is defined (which defaults to the.current environment). The function itself is fairly simple: it creates a call to `function` with the args and body as arguments, and then evaluates that in the correct environment so that the function has the right scope.

```
make_function <- function(args, body, env = parent.frame()) {
  args <- as.pairlist(args)

  eval(call("function", args, body), env)
}
```

(`pryr::make_function()` includes a little more error checking but is otherwise identical.)

Let's see a simple example

```
add <- make_function(alist(a = 1, b = 2), quote(a + b))
add(1)
#> [1] 3
add(1, 2)
#> [1] 3
```

Note our use of the `alist()` (**ar**gument list) function. We used this earlier when capturing unevaluated `...`, and we use it again here. Note that `alist()` doesn't evaluate its arguments and supports arguments with and without defaults (although if you don't want a default you need to be explicit). There's one small trick if you want to have `...` in the argument list: you need to use it on the left-hand side of an equals sign.

```
make_function(alist(a = , b = a), quote(a + b))
#> function (a, b = a)
#> a + b
#> <environment: 0x20c5608>
make_function(alist(a = , b = ), quote(a + b))
#> function (a, b)
#> a + b
#> <environment: 0x20c5608>
make_function(alist(a = , b = , ... =), quote(a + b))
#> function (a, b, ...)
#> a + b
#> <environment: 0x20c5608>
```

If you want to mix evaluated and unevaluated arguments, it might be easier to make the list by hand:

```
x <- 1
args <- list()
args$a <- x
args$b <- quote(expr = )

make_function(args, quote(a + b))
#> function (a = 1, b)
#> a + b
#> <environment: 0x20c5608>
```

Unenclose

Most of the time it's simpler to use closures to create new functions, but `make_function()` is useful if we want to make it obvious to the user what the function does (printing out a closure isn't usually that helpful because all the variables are present by name, not by value).

We could use `make_function()` to create an `unenclose()` function that takes a closure and modifies it so when you look at the source you can see what's going on:

```
unenclose <- function(f) {
  env <- environment(f)
  new_body <- substitute2(body(f), env)
  make_function(formals(f), new_body, parent.env(env))
}

f <- function(x) {
  function(y) x + y
}
f(1)
#> function(y) x + y
#> <environment: 0x13e37f30>
unenclose(f(1))
#> function (y)
#> 1 + y
#> <environment: 0x20c5608>
```

Exercises

- Why does `unenclose()` use `substitute2()`, not `substitute()`?

- Modify `unenclose` so it only substitutes in atomic vectors, not more complicated objects. (Hint: think about what the parent environment should be.)
- Read the documentation and source for `pryr::partial()` - what does it do? How does it work?

Walking the call tree with recursive functions

We've seen a couple of examples modifying a single call using `substitute()` or `modify_call()`. What if we want to do something more complicated, drilling down into a nested set of function calls and either extracting useful information or modifying the calls. The `codetools` package, included in the base distribution, provides some built-in tools for automated code inspection that use these ideas:

- `findGlobals()`: locates all global variables used by a function. This can be useful if you want to check that your functions don't inadvertently rely on variables defined in their parent environment.
- `checkUsage()`: checks for a range of common problems including unused local variables, unused parameters and use of partial argument matching.

In this section you'll learn how to write functions that do things like that.

Because code is a tree, we're going to need recursive functions to work with it. You now have basic understanding of how code in R is put together internally, and so we can now start to write some useful functions. The key to any function that works with the parse tree right is getting the recursion right, which means making sure that you know what the base case is (the leaves of the tree) and figuring out how to combine the results from the recursive case. The nodes of a tree are always calls (except in the rare case of function arguments, which are pairlists), and the leaves are names, single argument calls or constants. R provides a helpful function to distinguish whether an object is a node or a leaf: `is.recursive()`.

Finding F and T

We'll start with a function that returns a single logical value, indicating whether or not a function uses the logical abbreviations `T` and `F`. Using `T` and `F` is generally considered to be poor coding practice, and it's something that `R CMD check` will warn about.

When writing a recursive function, it's useful to first think about the simplest case: how do we tell if a leaf is a `T` or a `F`? This is very simple since the set of possibilities is small enough to enumerate explicitly:

```
is_logical_abbr <- function(x) {
  identical(x, quote(T)) || identical(x, quote(F))
}
is_logical_abbr(quote(T))
#> [1] TRUE
is_logical_abbr(quote(TRUE))
#> [1] FALSE
is_logical_abbr(quote(true))
#> [1] FALSE
is_logical_abbr(quote(10))
#> [1] FALSE
```

Next we write the recursive function. The base case is simple: if the object isn't recursive, then we just return the value of `is_logical_abbr()` applied to the object. If the object is not a node, then we work through each of the elements of the node in turn, recursively calling `logical_abbr()`. We need a special case for functions because we can't iterate through their components, instead we need to explicitly operate on the body and formals separately.

```
logical_abbr <- function(x) {
  # Base case
  if (!is.recursive(x)) return(is_logical_abbr(x))

  # Recursive cases
  if (is.function(x)) {
    if (logical_abbr(body(x))) return(TRUE)
    if (logical_abbr(formals(x))) return(TRUE)
  } else {
    for (i in seq_along(x)) {
      if (logical_abbr(x[[i]])) return(TRUE)
    }
  }

  FALSE
}

logical_abbr(quote(T))
#> [1] TRUE
logical_abbr(quote(mean(x, na.rm = T)))
#> [1] TRUE

f <- function(x = TRUE) {
  g(x + T)
}
logical_abbr(f)
#> [1] TRUE
```

Finding all variables created by assignment

In this section, we will write a function that figures out all variables that are created by assignment in an expression. We'll start simply, and make the function progressively more rigorous. One reason to start with this function is because the recursion is a little bit simpler - we never need to go all the way down to the leaves because we are looking for assignment, a call to `<-`.

This means that our base case is simple: if we're at a leaf, we've gone too far and can immediately return. We have two other cases: we have hit a call, in which case we should check if it's `<-`, otherwise it's some other recursive structure and we should call the function recursively on each element. Note the use of `identical()` to compare the call to the name of the assignment function, and recall that the second element of a call object is the first argument, which for `<-` is the left hand side: the object being assigned to.

```
is_call_to <- function(x, name) {
  is.call(x) && identical(x[[1]], as.name(name))
}

find_assign <- function(obj) {
  # Base case
  if (!is.recursive(obj)) return()

  if (is_call_to(obj, "<-")) {
    obj[[2]]
  } else {
    lapply(obj, find_assign)
  }
}
find_assign(quote(a <- 1))
#> a
find_assign(quote({
  a <- 1
  b <- 2
}))
#> [[1]]
#> NULL
#>
#> [[2]]
#> a
#>
#> [[3]]
#> b
```

This function seems to work for these simple cases, but the output is rather verbose. Instead of returning a list, let's keep it simple and stick with a character vector. We'll also test it with two slightly more complicated examples:

```
find_assign <- function(obj) {
  # Base case
  if (!is.recursive(obj)) return(character())

  if (is_call_to(obj, "<-")) {
    as.character(obj[[2]])
  } else {
    unlist(lapply(obj, find_assign))
  }
}
find_assign(quote({
  a <- 1
  b <- 2
  a <- 3
}))
#> [1] "a" "b" "a"

find_assign(quote({
  system.time(x <- print(y <- 5))
}))
#> [1] "x"
```

This is better, but we have two problems: repeated names, and we miss assignments inside function calls. The fix for the first problem is easy: we need to wrap `unique()` around the recursive case to remove duplicate assignments. The second problem is a bit more subtle: it's possible to do assignment within the arguments to a call, but we're failing to recurse down in to this case.

```
find_assign <- function(obj) {
  # Base case
  if (!is.recursive(obj)) return(character())

  if (is_call_to(obj, "<-")) {
    call <- as.character(obj[[2]])
    c(call, unlist(lapply(obj[[3]], find_assign)))
  } else {
    unique(unlist(lapply(obj, find_assign)))
  }
}
find_assign(quote({
  a <- 1
  b <- 2
  a <- 3
}))
#> [1] "a" "b"

find_assign(quote({
  system.time(x <- print(y <- 5))
}))
#> [1] "x" "y"
```

There's one more case we need to test:

```
find_assign(quote({
  ls <- list()
  ls$a <- 5
  names(ls) <- "b"
}))
#> [1] "ls"    "$"     "a"     "names"

call_tree(quote({
  ls <- list()
  ls$a <- 5
  names(ls) <- "b"
}))
#> \- {()
#>   \- <-()
#>     \- `ls
#>     \- list()
#>   \- <-()
#>     \- $()
#>       \- `ls
#>       \- `a
#>     \- 5
#>   \- <-()
#>     \- names()
#>       \- `ls
#>     \- "b"
```

This behaviour might be ok, but we probably just want assignment into whole objects, not assignment that modifies some property of the object. Drawing the tree for that quoted object helps us see what condition we should test for - we want the object on the left hand side of assignment to be a name. This gives the final version of the `find_assign` function.

```
find_assign <- function(obj) {
  # Base case
  if (!is.recursive(obj)) return(character())

  if (is_call_to(obj, "<-")) {
    call <- if (is.name(obj[[2]])) as.character(obj[[2]])
    c(call, unlist(lapply(obj[[3]], find_assign)))
  } else {
    unique(unlist(lapply(obj, find_assign)))
  }
}
find_assign(quote({
  ls <- list()
  ls$a <- 5
  names(ls) <- "b"
}))
#> [1] "ls"
```

Making this function work absolutely correct requires quite a lot more work, because we need to figure out all the other ways that assignment might happen: with `=`, `assign()`, or `delayedAssign()`. But a static tool can never be perfect: the best you can hope for is a set of heuristics that catches the most common 90% of cases.

Modifying the call tree

Instead of returning vectors computed from the contents of an expression, you can also return a modified expression, such as base R's `bquote()`. `bquote()` is a slightly more flexible form of quote: it allows you to optionally quote and unquote some parts of an expression (it's similar to the backtick operator in Lisp). Everything is quoted, *unless* it's encapsulated in `.()` in which case it's evaluated and the result is inserted.

```
a <- 1
b <- 3
bquote(a + b)
#> a + b
bquote(a + .(b))
#> a + 3
bquote(.(a) + .(b))
#> 1 + 3
bquote(.(a + b))
#> [1] 4
```

This provides a fairly easy way to control what gets evaluated when you call `bquote()`, and what gets evaluated when the expression is evaluated. How does `bquote()` work? Below, I've rewritten `bquote()` to use the same style as our other functions: it expects input to be quoted already, and makes the base and recursive cases more explicit:

```
bquote2 <- function (x, where = parent.frame()) {
  # Base case
  if (!is.recursive(x)) return(x)

  if (is.call(x)) {
    if (identical(x[[1]], quote(.))) {
      # Call to .(), so evaluate
      eval(x[[2]], where)
    } else {
      as.call(lapply(x, bquote2, where = where))
    }
  } else if (is.pairlist(x)) {
    as.pairlist(lapply(x, bquote2, where = where))
  } else {
    stop("Unknown case")
  }
}
x <- 1
bquote2(quote(x == .(x)))
#> x == 1
y <- 2
bquote2(quote(function(x = .(x)) {
  x + .(y)
}))
#> function(x = 1) {
#>     x + 2
#> }
```

Note that functions that modify the source tree are most useful for creating expressions that are used at run-time, not saved back into the original source file. That's because all non-code information is lost:

```
bquote2(quote(function(x = .(x)) {
  # This is a comment
  x +  # funky spacing
    .(y)
}))
#> function(x = 1) {
#>     x + 2
#> }
```

It is possible to work around this problem using `srcrefs` and `getParseData`, but neither solution naturally fits this hierarchical framework. You effectively end up having to recreate huge chunks of R's internal code in order to handle the majority of R code. So the above approach can be useful in simple cases (particularly when you don't care what the output code looks like), but it's very hard to automatically transform R code, and is beyond the scope of this book.

`bquote()` is rather like a macro from a languages like Lisp. But unlike macros the modifications occur at runtime, not compile time (which doesn't have any meaning in R). And unlike a macro there is no restriction to return an expression: a macro-like function in R can return anything. More like `fexprs`. a fexpr is like a function where the arguments aren't evaluated by default; or a macro where the result is a value, not code.

Programmer's Niche: Macros in R[3] by Thomas Lumley.

Exercises

- Write a function that extracts all calls to a function. Compare your function to `pryr::fun_calls()`.

Formulas

There is one other important tool of non-standard evaluation: the formula. The formula operator, `~`, is used extensively by modelling functions, but also by some graphics functions (e.g. lattice and `plot`) and a few data manipulation functions (e.g. `xtabs()` and `aggregate()`).

Formula as a quoting function

There are two advantages for using `~` over `quote()`:

- It is shorter
- It captures both the expression and the environment in which it was evaluated

The disadvantage of using `~` is that most people are used to its role in models, and may be surprised in the semantics you imply from it are substantially different from standard modelling formulas.

The formula object is a call that knows in which environment it was evaluated. You can use `length()` to determine if it is one-sided or two-sided, and `[[` to extract the various pieces.

[3] http://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf#page=11

```
f1 <- ~ a + b
length(f1)
f1[[1]]
f1[[2]]

f2 <- y ~ a + b
length(f2)
f2[[1]]
f2[[2]]
f2[[3]]
```

You can extract the environment of a formula with `environment()`, as demonstrated with this implementation of `subset()`:

```
subset_f <- function(x, f) {
  stopifnot(inherits(f, "formula"), length(f) == 2)
  r <- eval(f[[2]], x, environment(f))
  x[r, ]
}
subset_f(mtcars, ~ cyl == 4)
```

Note that because the code is evaluated in the environment associated with the formula, the semantics are a little different if you're creating the formula in a function:

```
f <- function(x) ~ cyl == x
subset_f(mtcars, f(4))
```

`xtabs()`

Is a pretty horrible example because of it's combination of call mangling and tangles with sparse matrices.

Formulas for modelling

Keep it brief: focus on main concepts (possibly showing complete lm implmentation using Rcpp), and pointing to documentation where necessary. Need to discuss specials (e.g. offset/Error) and how splines work?

White book.

Patsy: http://patsy.readthedocs.org/en/latest/R-comparison.html

Formula package (http://cran.r-project.org/web/packages/Formula/vignettes/Formula.pdf)

Models use two steps: first converting the formula into matrices, and then manipulating using matrix algebra.

- RcppEigen:::fastLm.formula
- http://developer.r-project.org/model-fitting-functions.txt
- terms, terms.object, terms.formula
- model.response, model.weights
- model.matrix, model.frame
- lm.fit

Chapter 14

Special contexts

R's lexical scoping rules, lazy argument evaluation and first-class environments make it an excellent environment for designing special environments that allow you to create domain specific languages (DSLs). This chapter shows how you can use these ideas to evaluate code in special contexts that pervert the usual behaviour of R. We'll start with simple modifications then work our way up to evaluations that completely redefine ordinary function semantics in R.

This chapter uses ideas from the breadth of the book including non-standard evaluation, environment manipulation, active bindings, …

- Evaluate code in a special context: `local`, `capture.output`, `with_*`
- Supply an expression instead of a function: `curve`
- Combine evaluation with extra processing: `test_that`, `assert_that`
- Create a full-blown DSL: `html`, `plotmath`, `deriv`, `parseNamespace`, `sql`

Evaluate code in a special context

It's often useful to evaluate a chunk of code in a special context

- Temporarily modifying global state, like the working directory, environment variables, plot options or locale.
- Capture side-effects of a function, like the text it prints, or the warnings it emits
- Evaluate code in a new environment

`with_something`

There are a number of parameters in R that have global state (e.g. `option()`, environmental variables, `par()`, ...) and it's useful to be able to run code temporarily in a different context. For example, it's often useful to be able to run code with the working directory temporarily set to a new location:

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

  force(code)
}
getwd()
in_dir("~", getwd())
```

The basic pattern is simple:

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

We could use similar code to temporarily set global options:

```
with_options <- function(opts, code) {
  old <- options(opts)
  on.exit(options(old))

  force(code)
}
x <- 0.123456
print(x)
with_options(list(digits = 3), print(x))
with_options(list(digits = 1), print(x))
```

You might notice that there's a lot of similarity between these two functions. We could extract out that commonality with a function operator that takes the setter function as an input:

```
with_something <- function(set) {
  function(new, code) {
    old <- set(new)
    on.exit(set(old))
    force(code)
  }
}
```

Then we can easily generate a whole set of with functions:

```
in_dir <- with_something(setwd)
with_options <- with_something(options)
with_par <- with_something(par)
```

However, many of the setter functions that affect global state in R don't return the previous values in a way that can easily be passed back in. In that case, like for `.libPaths()`, which controls where R looks for packages to load, we first create a wrapper that enforces the behaviour we want, and then use `with_something()`:

```
set_libpaths <- function(paths) {
  libpath <- normalizePath(paths, mustWork = TRUE)

  old <- .libPaths()
  .libPaths(paths)
  invisible(old)
}
with_libpaths <- with_something(set_libpaths)
```

These functions (and a few others) are provided by the devtools package. See `?with_something` for more details.

`capture.output`

`capture.output()` is a useful function when the output you really want from a function is printed to the console. It also allows you to work around badly written functions that have no way to suppress output to the console. For example, it's difficult to capture the output of `str()`:

```
y <- 1:10
y_str <- str(y)
y_str

y_str <- capture.output(str(y))
y_str
```

To work its magic, `capture.output()` uses `sink()`, which allows you to redirect the output stream to an arbitrary connection. We'll first write a helper function that allows us to execute code in the context of a `sink()`, automatically un-`sink()`ing when the function finishes:

```
with_sink <- function(connection, code, ...) {
  sink(connection, ...)
  on.exit(sink())

  code
}
with_sink("temp.txt", print("Hello"))
readLines("temp.txt")
```

With this in hand, we just need to add a little extra wrapping to our `capture.output2()` to write to a temporary file, read from it and clean up after ourselves:

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

  with_sink(temp, force(code))
  readLines(temp)
}
capture.output2(cat("a", "b", "c", sep = "\n"))
```

The real `capture.output()` is a bit more complicated: it uses a local `textConnection` to capture the data sent to sink, and it allows you to supply multiple expressions which are evaluated in turn. Using `with_sink()` this looks like `capture.output3()`

```
capture.output3 <- function(..., env = parent.frame()) {
  txtcon <- textConnection("rval", "w", local = TRUE)

  with_sink(txtcon, {
    args <- dots(...)
    for(i in seq_along(args)) {
      out <- withVisible(eval(args[[i]], env))
      if (out$visible) print(out$value)
    }
  })

  rval
}
```

You might want to compare this function to the real `capture.output()` and think about the simplifications I've made. Is the code easier to understand or harder? Have I removed important functionality?

If you want to capture more types of output (like messages and warnings), you may find the `evaluate` package helpful. It powers `knitr`, and does it's best to ensure high fidelity between its output and what you'd see if you copy and pasted the code at the console.

Evaluating code in a new environment

In the process of performing a data analysis, you may create variables that are necessarily because they help break a complicated sequence of steps down in to easily digestible chunks, but are not needed afterwards. For example, in the following example, we might only want to keep the value of c:

```
a <- 10
b <- 30
c <- a + b
```

It's useful to be able to store only the final result, preventing the intermediate results from cluttering your workspace. We already know one way of doing this, using a function:

```
c <- (function() {
  a <- 10
  b <- 30
  a + b
})()
```

(In JavaScript this is called the immediately invoked function expression (IIFE), and is used extensively in modern JavaScript to encapsulate different JavaScript libraries)

R provides another tool that's a little less verbose, the `local()` function:

```
c <- local({
  a <- 10
  b <- 30
  a + b
})
```

The idea of local is to create a new environment (inheriting from the current environment) and run the code in that. The essence of `local()` is captured in this code:

```
local2 <- function(expr) {
  envir <- new.env(parent = parent.frame())
  eval(substitute(expr), envir)
}
```

The real `local()` code is considerably more complicated because it adds a second environment parameter. I don't think this is necessary because if you have an explicit environment parameter, then you can already evaluate code in that environment with `evalq()`. The original code is also hard to understand because it is very concise and uses some sutble features of evaluation (including non-standard evaluation of both arguments). If you have read [[computing-on-the-language]], you might be able to puzzle it out, but to make it a bit easier I have rewritten it in a simpler style below.

```
local2 <- function(expr, envir = new.env()) {
  env <- parent.frame()
  call <- substitute(eval(quote(expr), envir))

  eval(call, env)
}
a <- 100
local2({
  b <- a + sample(10, 1)
  my_get <<- function() b
})
my_get()
```

You might wonder we can't simplify to this:

```
local3 <- function(expr, envir = new.env()) {
  eval(substitute(expr), envir)
}
```

But it's because of how the arguments are evaluated - default arguments are evalauted in the scope of the function so that `local(x)` would not the same as `local(x, new.env())` without special effort.

Exercises

- Compare `capture.output()` to `capture.output2()` and `local()` to `local2()`. 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?
- How does the `chdir` parameter of `source()` compare to `in_dir()`? Why might you prefer one approach to the other?

Anaphoric functions

Another variant along these lines is an "anaphoric[1] function", or a function that uses a pronoun. This is easiest to understand with an example using an interesting anaphoric function in base R: `curve()`.`curve()` draws a plot of the specified function, but interestingly you don't need to use a function, you just supply an expression that uses `x`:

```
curve(x ^ 2)
curve(sin(x), to = 3 * pi)
curve(sin(exp(4 * x)), n = 1000)
```

Here `x` plays a role like a pronoun in an English sentence: it doesn't represent a single concrete value, but instead is a place holder that varies over the range of the plot. Note that it doesn't matter what the value of `x` outside of `curve()` is: the expression is evaluated in a special environment where `x` has a special meaning:

```
x <- 1
curve(sin(exp(4 * x)), n = 1000)
```

`curve()` works by evaluating the expression in a special environment in which the appropriate `x` exists. The essence of `curve()`, omitting many useful but incidental details like plot labelling, looks like this:

```
curve2 <- function(expr, xlim = c(0, 1), n = 100, env = parent.frame()) {
  env2 <- new.env(parent = env)
  env2$x <- seq(xlim[1], xlim[2], length = n)

  y <- eval(expr, env2)
  plot(x, y, type = "l", ylab = deparse(substitute(expr)))
}
curve2(sin(exp(4 * x)), n = 1000)
```

Creating a new environment containing the pronoun is the key technique for implementing anaphoric functions.

Another way to solve the problem would be to turn the expression into a function using `make_function()`:

```
curve3 <- function(expr, xlim, n = 100, env = parent.frame()) {
  f <- make_function(alist(x = ), substitute(expr), env)
```

[1] http://en.wikipedia.org/wiki/Anaphora_linguistics

```
  x <- seq(xlim[1], xlim[2], length = n)
  y <- f(x)

  plot(x, y, type = "l", ylab = deparse(substitute(expr)))
}
curve3(sin(exp(4 * x)), n = 1000)
```

The approaches take about as much code, and require knowledge of the same number of fundamental R concepts. I would have a slight preference for the second because it would be easier to reuse the part of the `curve3()` that turns an expression into a function. All anaphoric functions need careful documentation so that the user knows that some variable will have special properties inside the anaphoric function and must otherwise be avoided.

If you're interesting in learning more, there are some good resources for anaphoric functions in Arc[2] (a list like language), Perl[3] and Clojure[4]

With connection

We could use this idea to create a function that allows you to use a connection, automatically openning and closing it if needed. This is a common technique in Ruby, which uses its block syntax whenever you deal with resources that need to be closed after you're done with them. We can't use the same implementation as the with functions above because an implementation like that below gives no way for the code to refer to the connection:

```
with_conn <- function(conn, code) {
  open(conn)
  on.exit(close(conn))

  force(code)
}
```

We can work around this problem by using an anaphoric function, referring to the connection as `.it`. We'd need to clearly document that convention so the user knew what to call it.

```
with_conn <- function(conn, code, env = parent.frame()) {
  if (!isOpen(conn)) {
    open(conn)
```

[2]http://www.arcfn.com/doc/anaphoric.html
[3]http://www.perlmonks.org/index.pl?node_id=666047
[4]http://amalloy.hubpages.com/hub/Unhygenic-anaphoric-Clojure-macros-for-fun-and-profit

```
    on.exit(close(conn))
  }

  env2 <- new.env(parent = env)
  env2$.it <- conn

  eval(substitute(code), env2)
}

# Create a new file, and then read it in in pieces of length 6
with_conn(file("test.txt", "w+"), {
  writeLines("This is a test", .it)
  x <- readChar(.it, 6)

  while(length(x) > 0) {
    print(x)
    x <- readChar(.it, 6)
  }
})
```

The construction of this function also has the appealing side-effect that the scope of the connection is clearly shown with indenting.

Special environments for run-time checking

We can take the idea of special evaluation contexts and use the idea to implement run-time checks, by executing code in a special environment that warns or errors when the user does something that we think is a bad idea.

- Checking for logical abbreviations

Logical abbreviations

One of the checks that `R CMD check` runs ensures that you don't use the logical abbreviations `T` and `F`. It's a useful check, but a bit frustrating to use because it terminates on the first `F` or `T` it finds, so you need to run it many times, but `R CMD check` is slow. Instead, we can take the idea that underlies the check and run it ourselves, which makes it faster to iterate.

The basic idea is to use an active binding so that whenever code tries to access `F` or `T` it throws an error. This is a very similar technique to anaphoric functions:

```
check_logical_abbr <- function(code, env = parent.frame()) {
  new_env <- new.env(parent = env)
  delayedAssign("T", stop("Use TRUE not T"), assign.env = new_env)
  delayedAssign("F", stop("Use FALSE not F"), assign.env = new_env)

  eval(substitute(code), new_env)
}

check_logical_abbr(c(FALSE, T, FALSE))
```

Note that functions look in the environment in which they were defined so to test large bodies of code, you'll need to run `check_logical_abbr()` as far as out as possible:

```
f <- function(x) {
  mean(x, na.rm = T)
}
check_logical_abbr(f(1:10))

check_logical_abbr({
  f <- function(x) {
    mean(x, na.rm = T)
  }
  f(1:10)
})
```

This will be typically easiest to do as a wrapper around the code you use to load your code into R:

```
check_logical_abbr(source("my-file.r"))
check_logical_abbr(load_all())
```

Test that

Manage global state accessible from functions. Use special environment.

Create with function that uses `on.exit()` to push and pop from stack.

```
testthat_env <- new.env()
testthat_env$reporter <- StopReporter$new()

set_reporter <<- function(value) {
  old <- testthat_env$reporter
  testthat_env$reporter <- value
  old
}
```

```
get_reporter <<- function() {
  testthat_env$reporter
}

with_reporter <- function(reporter, code) {
  reporter <- find_reporter(reporter)

  old <- set_reporter(reporter)
  on.exit(set_reporter(old))

  reporter$start_reporter()
  res <- force(code)
  reporter$end_reporter()

  res
}
```

Chapter 15

Domain specific languages

The combination of first class environments and lexical scoping gives us a powerful toolkit for creating embedded domain specific languages (DSLs) in R. Embedded DSLs take advantage of a host language's parsing and execution framework, but adjust the semantics somewhat to make them more suitable for a specific task.

R already has a simple and popular DSL built in: the formula specification, which offers a succinct way of describing the relationship between predictors and the response. Other examples of DSLs include ggplot2 (for visualisation), and plyr (for data manipulation). Another package that makes extensive use of these ideas is dplyr, which provides `translate_sql()` to convert R expressions into SQL:

```
library(dplyr)
translate_sql(sin(x) + tan(y))
#> <SQL> SIN("x") + TAN("y")
translate_sql(x < 5 & !(y >= 5))
#> <SQL> "x" < 5.0 AND NOT(("y" >= 5.0))
translate_sql(first %like% "Had*")
#> <SQL> "first" LIKE 'Had*'
translate_sql(first %in% c("John", "Roger", "Robert"))
#> <SQL> "first" IN ('John', 'Roger', 'Robert')
translate_sql(like == 7)
#> <SQL> "like" = 7.0
```

Once you have read this chapter, you might want to study the source code for dplyr. An important part of the overall structure of the package is `partial_eval()` which helps manage expressions where some of the components refer to variables in the database and some refer to local R objects. You could use very similar ideas if you needed to translate small R expressions into other languages, like JavaScript or Python. Converting complete R programs would be extremely difficult, but often being able to communicate a simple description of computation between languages is very useful.

R is well suited for hosting DSLs because the combination of a small amount of computing on the language and constructing special evaluation environments is very powerful. Creating new DSLs in R uses many techniques that you've learned about elsewhere in the book, including:

- scoping rules
- creating and manipulating functions
- computing on the language
- S3 basics

This chapter will develop two simple, but useful, DSLs, one for generating HTML, and one for turning R mathematical expressions into a form suitable for inclusion in LaTeX.

DSLs are a very large topic, and this chapter will only scratch the surface, focussing on important techniques and not so much on how you might come up with the language in the first place. If you're interested in learning more, I highly recommend Domain Specific Languages[1] by Martin Fowler: it discusses many options for creating a DSL and provides many examples of different languages.

HTML

HTML is the language that underlies the majority of the web. It is a special case of SGML, and similar (but not identical) to XML. HTML looks like this:

```html
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

Even if you've never seen HTML before, hopefully you can see the key component of the structure: HTML is composed of tags that look like `<tag></tag>`. Tags can be contained inside other tags and intermingled with text. Generally, HTML ignores whitespace: an sequence of whitespace is equivalent to a single space. You could put the previous example all on one line and it would still display the same in the browser:

[1] http://amzn.com/0321712943?tag=devtools-20

```
<body><h1 id='first'>A heading</h1><p>Some text &amp; <b>some bold
text.</b></p><img src='myimg.png' width='100' height='100' />
</body>
```

However, like R code, you usually want to indent HTML to make it more obvious to see the structure.

There are over 100 HTML tags, but to illustrate HTML we're going to focus on just a few:

- `<body>`: the top-level tag that all content is enclosed within
- `<h1>`: creates a heading-1, the top level heading
- `<p>`: creates a paragraph
- `<b>`: emboldens text
- `<img>`: embeds an image

(you probably guessed what these did already!)

Tags can also have named attributes that look like `<tag a="a" b="b"></tags>`. Tag values should always be enclosed in either single or double quotes. Two important attributes used on just about every tag are `id` and `class`. These are used in conjunction with CSS (cascading style sheets) in order to control the style of the document.

Some tags, like `<img>`, can't have any content. These are called **void tags** and have a slightly different syntax: instead of writing `<img></img>` you write `<img />`. Since they have no content, attributes are more imporant, and `img` has three that are used for almost every image: `src` (where the image lives), `width` and `height`.

Because `<` and `>` have special meanings in HTML, you can't write them directly. Instead you have to use the HTML escapes `&gt;` and `&lt;`. And since those escapes use `&`, you also have to escape it with `&amp;` if you want a literal ampersand.

Goal

Our goal is to make it easy to generate HTML from R. To give a concrete example, we want to generate the following HTML:

```
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100' />
</body>
```

using code that looks as similar to the HTML as possible. We will work our way up to the following DSL:

```r
with_html(body(
  h1("A heading", id = "first"),
  p("Some text &", b("some bold text.")),
  img(src = "myimg.png", width = 100, height = 100)
))
```

Note that the nesting of function calls is the same as the nesting of tags, unnamed arguments become the content of the tag, and named arguments become the attributes. Because tags and text are clearly distinct in this API, we can automatically escape & and other special characters.

Escaping

Escaping is so fundamental we're going to start with it. We first start by creating a way of escaping the characters that have special meaning for HTML, while making sure we don't end up double-escaping at any point. The easiest way to do this is to create an S3 class that allows us to distinguish between regular text (that needs escaping) and HTML (that doesn't).

We then write an escape method that leaves HTML unchanged and escapes the special characters (&, <, >) in ordinary text. We also add a method for lists for convenience

```r
html <- function(x) structure(x, class = "html")
print.html <- function(x, ...) cat("<HTML> ", x, "\n", sep = "")
escape <- function(x) UseMethod("escape")
escape.html <- function(x) x
escape.character <- function(x) {
  x <- gsub("&", "&amp;", x)
  x <- gsub("<", "&lt;", x)
  x <- gsub(">", "&gt;", x)

  html(x)
}
escape.list <- function(x) {
  lapply(x, escape.character)
}

# Now we check that it works
escape("This is some text.")
#> [1] "This is some text."
#> attr(,"class")
#> [1] "html"
escape("x > 1 & y < 2")
#> [1] "x &gt; 1 &amp; y &lt; 2"
#> attr(,"class")
#> [1] "html"

# Double escaping is not a problem
escape(escape("This is some text. 1 > 2"))
#> [1] "This is some text. 1 &gt; 2"
#> attr(,"class")
#> [1] "html"

# And text we know is HTML doesn't get escaped.
escape(html("<hr />"))
#> [1] "<hr />"
#> attr(,"class")
#> [1] "html"
```

Escaping is an important component for any DSL.

Basic tag functions

Next we'll write a few simple tag functions and then figure out how to generalise for all possible HTML tags. Let's start with `<p>`. HTML tags can have both attributes (e.g. id, or class) and children (like `<b>` or `<i>`). We need some way of separating these in the function call: since attributes are named values and children don't have names, it seems natural to separate using named vs. unnamed arguments. Then a call to `p()` might look like:

```
p("Some text.", b("some bold text"), class = "mypara")
```

We could list all the possible attributes of the p tag in the function definition, but that's hard because there are so many, and it's possible to use custom attributes[2] Instead we'll just use ... and separate the components based on whether or not they are named. To do this correctly, we need to be aware of a "feature" of `names()`:

```
names(c(a = 1, b = 2))
#> [1] "a" "b"
names(c(a = 1, 2))
#> [1] "a" ""
names(c(1, 2))
#> NULL
```

[2] http://html5doctor.com/html5-custom-data-attributes/

With this in mind we create two helper functions to extract the named and unnamed components of a vector:

```
named <- function(x) {
  if (is.null(names(x))) return(NULL)
  x[names(x) != ""]
}
unnamed <- function(x) {
  if (is.null(names(x))) return(x)
  x[names(x) == ""]
}
```

We can now create our `p()` function. There's one new function here: `html_attributes()`. This takes a list of name-value pairs and creates the correct HTML attributes specification from them. It's a little complicated (to deal with some idiosyncracies of HTML that I haven't mentioned), not that important and doesn't introduce any new ideas, so I won't discuss it here, but it's included at the end of the chapter.

```
source("code/html-attributes.r", local = TRUE)
p <- function(...) {
  args <- list(...)
  attribs <- html_attributes(named(args))
  children <- unlist(escape(unnamed(args)))

  html(paste0(
    "<p", attribs, ">",
    paste(children, collapse = ""),
    "</p>"
  ))
}

p("Some text")
#> [1] "<p>Some text</p>"
#> attr(,"class")
#> [1] "html"
p("Some text", id = "myid")
#> [1] "<p id = 'myid'>Some text</p>"
#> attr(,"class")
#> [1] "html"
p("Some text", image = NULL)
#> [1] "<p image>Some text</p>"
#> attr(,"class")
#> [1] "html"
p("Some text", class = "important", "data-value" = 10)
```

```
#> [1] "<p class = 'important' data-value = '10'>Some text</p>"
#> attr(,"class")
#> [1] "html"
```

Tag functions

With this definition of `p()` it's pretty easy to see what will change for different tags: we just need to replace `"p"` with a variable. We'll use a closure to make it easy to generate a tag function given a tag name:

```
tag <- function(tag) {
  force(tag)
  function(...) {
    args <- list(...)
    attribs <- html_attributes(named(args))
    children <- unlist(escape(unnamed(args)))

    html(paste0(
      "<", tag, attribs, ">",
      paste(children, collapse = ""),
      "</", tag, ">"
    ))
  }
}
```

(We're forcing the evaluation `tag` with the expectation we'll be calling this function from a loop later on - that avoids potential bugs caused by lazy evaluation.)

Now we can run our earlier example:

```
p <- tag("p")
b <- tag("b")
i <- tag("i")
p("Some text.", b("Some bold text"), i("Some italic text"),
  class = "mypara")
#> [1] "<p class = 'mypara'>Some text.&lt;b&gt;Some bold text&lt;/b&gt;&lt;i&gt;Some ita
#> attr(,"class")
#> [1] "html"
```

Before we continue to generate functions for every possible HTML tag, we need a variant of `tag()` for void tags. It can be very similar to `tag()`, but needs to throw an error if there are any unnamed tags, and the tag itself looks slightly different:

```
void_tag <- function(tag) {
  force(tag)
  function(...) {
    args <- list(...)
    if (length(unnamed(args)) > 0) {
      stop("Tag ", tag, " can not have children", call. = FALSE)
    }
    attribs <- html_attributes(named(args))

    html(paste0("<", tag, attribs, " />"))
  }
}

img <- void_tag("img")
img(src = "myimage.png", width = 100, height = 100)
#> [1] "<img src = 'myimage.png' width = '100' height = '100' />"
#> attr(,"class")
#> [1] "html"
```

Processing all tags

Next we need a list of all the HTML tags:

```
tags <- c("a", "abbr", "address", "article", "aside", "audio", "b",
  "bdi", "bdo", "blockquote", "body", "button", "canvas", "caption",
  "cite", "code", "colgroup", "data", "datalist", "dd", "del",
  "details", "dfn", "div", "dl", "dt", "em", "eventsource",
  "fieldset", "figcaption", "figure", "footer", "form", "h1", "h2",
  "h3", "h4", "h5", "h6", "head", "header", "hgroup", "html", "i",
  "iframe", "ins", "kbd", "label", "legend", "li", "mark", "map",
  "menu", "meter", "nav", "noscript", "object", "ol", "optgroup",
  "option", "output", "p", "pre", "progress", "q", "ruby", "rp",
  "rt", "s", "samp", "script", "section", "select", "small", "span",
  "strong", "style", "sub", "summary", "sup", "table", "tbody",
  "td", "textarea", "tfoot", "th", "thead", "time", "title", "tr",
  "u", "ul", "var", "video")

void_tags <- c("area", "base", "br", "col", "command", "embed",
  "hr", "img", "input", "keygen", "link", "meta", "param", "source",
  "track", "wbr")
```

If you look at this list carefully, you'll see there are quite a few tags that have the same name as base R functions (`body`, `col`, `q`, `source`, `sub`, `summary`, `table`), and others that clash with popular packages (e.g. `map`). That implies we don't want to make all the functions available (in either the global environment or a package environment) by default. Instead, we'll put them in a list, and add some additional code to make it easy to use them when desired. First we make a named list:

```
tag_fs <- c(
  setNames(lapply(tags, tag), tags),
  setNames(lapply(void_tags, void_tag), void_tags)
)
```

This gives us a way to call tag functions explicitly, but is a little verbose:

```
tag_fs$p("Some text.", tag_fs$b("Some bold text"),
  tag_fs$i("Some italic text"))
#> [1] "<p>Some text.&lt;b&gt;Some bold text&lt;/b&gt;&lt;i&gt;Some italic text&lt;/i&gt
#> attr(,"class")
#> [1] "html"
```

Then we finish off our HTML DSL by creating a function that allows us to evaluate code in the context of that list:

```
with_html <- function(code) {
  eval(substitute(code), tag_fs)
}
```

This gives us a succinct API which allows us to write HTML when we need it without cluttering up the namespace when we don't. Inside `with_html` if you want to access the R function overridden by an HTML tag of the same name, you can use the full `package::function` specification.

```
with_html(body(
  h1("A heading", id = "first"),
  p("Some text &", b("some bold text.")),
  img(src = "myimg.png", width = 100, height = 100)
))
#> [1] "<body>&lt;h1 id = 'first'&gt;A heading&lt;/h1&gt;&lt;p&gt;Some text &amp;amp;&am
#> attr(,"class")
#> [1] "html"
```

Exercises

- The escaping rules for `<script>` and `<style>` tags are different: you don't want to escape angle brackets or ampersands, but you do want to escape </. Adapt the code above to follow these rules.

- The use of ... for all functions has some big downsides: there's no input validation and there will be little information in the documentation or autocomplete about how to use the function. Create a new function that when given a named list of tags and their attribute names (like below), creates functions with those signatures.

  ```
  list(
    a = c("href"),
    img = c("src", "width", "height")
  )
  ```

  All tags should get `class` and `id` attributes.

- Currently the HTML doesn't look terribly pretty, and it's hard to see the structure. How could you adapt `tag()` to do be indenting and formatting?

LaTeX

The next DSL we're going to tackle will convert R expression into their LaTeX math equivalents. (This is a bit like `?plotmath`, but for text instead of plots.) LaTeX is the lingua franca of mathematicians and statisticians: whenever you want to describe an equation in text (e.g. in an email) you write it as a LaTeX equation. Many reports are produced from R using LaTeX, so it might be useful to facilitate the automate conversion from mathematical expressions from one language to the other.

This math expression DSL will be more complicated than the HTML DSL, because not only do we need to convert functions, but we also need to convert symbols. We'll also create a "default" conversion, so that functions we don't know how to convert get a standard fallback. Like the HTML DSL, we'll also write functionals to make it easier to generate the translators.

Before we begin, let's quickly cover how formulas are expressed in LaTeX.

LaTeX mathematics

LaTeX mathematics are complex, and well documented[3]. They have a fairly simple structure:

- Most simple mathematical equations are represented in the way you'd type them into R: `x * y`, `z ^ 5`. Subscripts are written using `_`, e.g. `x_1`.

[3]http://en.wikibooks.org/wiki/LaTeX/Mathematics

- Special characters start with a \: `\pi` = , `\pm` = ±, and so on. There are a huge number of symbols available in LaTeX. Googling for `latex math symbols` finds many lists[4], and there's even a service[5] where you can sketch a symbol in the browser and it will look it up for you.
- More complicated functions look like `\name{arg1}{arg2}`. For example to represent a fraction you use `\frac{a}{b}`, and a sqrt looks like `\sqrt{a}`.
- To group elements together use `{}`: i.e. `x ^ a + b` vs. `x ^ {a + b}`.
- In good math typesetting, a distinction is made between variables and functions, but without extra information, LaTeX doesn't know whether `f(a * b)` represents calling the function `f` with argument `a * b`, or is shorthand for `f * a * b`. If `f` is a function, you can tell LaTeX to typeset it using an upright font with `\textrm{f}(a * b)`

Goal

Our goal is to use these rules to automatically convert from an R expression to a LaTeX representation of that expression. We will tackle it in four stages:

- Convert known symbols: `pi` -> `\pi`
- Leave other symbols unchanged: `x` -> `x`, `y` -> `y`
- Convert known functions: `x * pi` -> `x * \pi`, `sqrt(frac(a, b))` -> `\sqrt{\frac{a, b}}`
- Wrap unknown functions with `\textrm`: `f(a)` -> `\textrm{f}(a)`

Compared to the HTML DSL, we'll work in the opposite direction: we'll start with the infrastructure and work our way down to generate all the functions we need

`to_math`

To begin, we need a wrapper function that we'll use to convert R expressions into LaTeX math expressions. This works the same way as `to_html`: we capture the unevaluated expression and evaluate it in a special environment. However, the special environment is no longer fixed, and will vary depending on the expression. We need this in order to be able to deal with symbols and functions that we don't know about a priori.

```
to_math <- function(x) {
  expr <- substitute(x)
  eval(expr, latex_env(expr))
}
```

[4] http://www.sunilpatel.co.uk/latex-type/latex-math-symbols/
[5] http://detexify.kirelabs.org/classify.html

Known symbols

Our first step is to create an environment that allows us to convert the special LaTeX symbols used for Greek, e.g. `pi` to `\pi`. This is the same basic trick used in `subset` to make it possible to select column ranges by name (`subset(mtcars, , cyl:wt)`): we just bind a name to a string in a special environment.

First we create than environment by creating a named vector, converting that vector into a list, and then turn that list into an environment.

```
greek <- c(
  "alpha", "theta", "tau", "beta", "vartheta", "pi", "upsilon",
  "gamma", "gamma", "varpi", "phi", "delta", "kappa", "rho",
  "varphi", "epsilon", "lambda", "varrho", "chi", "varepsilon",
  "mu", "sigma", "psi", "zeta", "nu", "varsigma", "omega", "eta",
  "xi", "Gamma", "Lambda", "Sigma", "Psi", "Delta", "Xi", "Upsilon",
  "Omega", "Theta", "Pi", "Phi")
greek_list <- setNames(paste0("\\", greek), greek)
greek_env <- list2env(as.list(greek_list), parent = emptyenv())
```

We can then check it:

```
latex_env <- function(expr) {
  greek_env
}

to_math(pi)
#> [1] "\\pi"
to_math(beta)
#> [1] "\\beta"
```

Unknown symbols

If a symbol isn't greek, we want to leave it as is. This is trickier because we don't know in advance what symbols will be used, and we can't possibly generate them all. So we'll use a little bit of computing on the language to find out what symbols are present in an expression. The `all_names` function takes an expression: if it's a name, it converts it to a string; if it's a call, it recurses down through its arguments.

```
all_names <- function(x) {
  # Base cases
  if (is.name(x)) return(as.character(x))
  if (!is.call(x)) return(NULL)
  # Recursive case
  children <- lapply(x[-1], all_names)
  unique(unlist(children))
}

all_names(quote(x + y + f(a, b, c, 10)))
#> [1] "x" "y" "a" "b" "c"
# [1] "x" "y" "a" "b" "c"
```

We now want to take that list of symbols, and convert it to an environment so that each symbol is mapped to a string representing itself (e.g. so `eval(quote(x), env)` yields `"x"`). We again use the pattern of converting a named character vector to a list, then an environment.

```
latex_env <- function(expr) {
  names <- all_names(expr)
  symbol_list <- setNames(as.list(names), names)
  symbol_env <- list2env(symbol_list)

  symbol_env
}

to_math(x)
#> [1] "x"
to_math(longvariablename)
#> [1] "longvariablename"
to_math(pi)
#> [1] "pi"
```

This works, but we need to combine it with the enviroment of the Greek symbols. Since we want to prefer Greek to the defaults (e.g. `to_math(pi)` should give `"\\pi"`, not `"pi"`), `symbol_env` needs to be the parent of `greek_env`, and thus we need to make a copy of `greek_env` with a new parent. Strangely R doesn't come with a function for cloning environments, but we can easily create one by combining two existing functions:

```
clone_env <- function(env, parent = parent.env(env)) {
  list2env(as.list(env), parent = parent)
}
```

This gives us a function that can convert both known (Greek) and unknown symbols.

```
latex_env <- function(expr) {
  # Unknown symbols
  names <- all_names(expr)
  symbol_list <- setNames(as.list(names), names)
  symbol_env <- list2env(symbol_list)

  # Known symbols
  clone_env(greek_env, symbol_env)
}

to_math(x)
#> [1] "x"
to_math(longvariablename)
#> [1] "longvariablename"
to_math(pi)
#> [1] "\\pi"
```

Known functions

Next we'll add functions to our DSL. We'll start with a couple of helper closures that make it easy to add new unary and binary operators. These functions are very simple since they only have to assemble strings. (Again we use `force` to make sure the arguments are evaluated at the right time.)

```
unary_op <- function(left, right) {
  force(left)
  force(right)
  function(e1) {
    paste0(left, e1, right)
  }
}

binary_op <- function(sep) {
  force(sep)
  function(e1, e2) {
    paste0(e1, sep, e2)
  }
}
```

Using these helpers, we can map a few illustrative examples from R to LaTeX. Note how the lexical scoping rules of R help us: we can easily provide new meanings for standard functions like `+`, `-` and `*`, and even `(` and `{`.

```
# Binary operators
f_env <- new.env(parent = emptyenv())
```

```
f_env$"+" <- binary_op(" + ")
f_env$"-" <- binary_op(" - ")
f_env$"*" <- binary_op(" * ")
f_env$"/" <- binary_op(" / ")
f_env$"^" <- binary_op("^")
f_env$"[" <- binary_op("_")

# Grouping
f_env$"{" <- unary_op("\\left{ ", " \\right}")
f_env$"(" <- unary_op("\\left( ", " \\right)")
f_env$paste <- paste

# Other math functions
f_env$sqrt <- unary_op("\\sqrt{", "}")
f_env$sin <- unary_op("\\sin(", ")")
f_env$log <- unary_op("\\log(", ")")
f_env$abs <- unary_op("\\left| ", "\\right| ")
f_env$frac <- function(a, b) {
  paste0("\\frac{", a, "}{", b, "}")
}

# Labelling
f_env$hat <- unary_op("\\hat{", "}")
f_env$tilde <- unary_op("\\tilde{", "}")
```

We again modify `latex_env()` to include this environment. It should be the last environment in which names are looked for, so that `sin(sin)` works. (because of R's matching rules wrt functions vs. other objects)

```
latex_env <- function(expr) {
  # Known functions
  f_env

  # Default symbols
  names <- all_names(expr)
  symbol_list <- setNames(as.list(names), names)
  symbol_env <- list2env(symbol_list, parent = f_env)

  # Known symbols
  greek_env <- clone_env(greek_env, parent = symbol_env)
}

to_math(sin(x + pi))
#> [1] "\\sin(x + \\pi)"
to_math(log(x_i ^ 2))
```

```
#> [1] "\\log(x_i^2)"
to_math(sin(sin))
#> [1] "\\sin(sin)"
```

Unknown functions

Finally, we'll add a default for functions that we don't know about. Like the unknown names, we can't know in advance what these will be, so we again use a little computing on the language to figure them out:

```
all_calls <- function(x) {
  # Base name
  if (!is.call(x)) return(NULL)

  # Recursive case
  fname <- as.character(x[[1]])
  children <- lapply(x[-1], all_calls)
  unique(c(fname, unlist(children, use.names = FALSE)))
}

all_calls(quote(f(g + b, c, d(a))))
#> [1] "f" "+" "d"
```

And we need a closure that will generate the functions for each unknown call

```
unknown_op <- function(op) {
  force(op)
  function(...) {
    contents <- paste(..., collapse = ", ")
    paste0("\\mathrm{", op, "}(", contents, ")")
  }
}
```

And again we update `latex_env()`:

```
latex_env <- function(expr) {
  calls <- all_calls(expr)
  call_list <- setNames(lapply(calls, unknown_op), calls)
  call_env <- list2env(call_list)

  # Known functions
  f_env <- clone_env(f_env, call_env)
  # Default symbols
  symbols <- all_names(expr)
  symbol_list <- setNames(as.list(symbols), symbols)
  symbol_env <- list2env(symbol_list, parent = f_env)

  # Known symbols
  greek_env <- clone_env(greek_env, parent = symbol_env)
}

to_math(f(a * b))
#> [1] "\\mathrm{f}(a * b)"
```

Exercises

- Add automatic escaping. Special symbols that should be escaped by adding a backslash in front of them are `\`, `$` and `%`. Like for sql, you'll need to make sure you don't end up double-escaping, so you'll need to create a small s3 class and then use that in function operators. That will also allow you to embed arbitrary LaTeX if needed.
- Complete the DSL to support all the functions that `plotmath` supports
- There's a repeating pattern in `latex_env()`: we take a character vector, do something to each piece, then convert it to a list, and then an environment. Write a function to automate this task, and then rewrite `latex_env()`

Part IV

Performance

Chapter 16

Performance

General techniques for improving performance.

Find out what is slow. Then make it fast.

Micro-benchmarking

Once you have identified the performance bottleneck in your code, you'll want to try out many variant approaches.

The microbenchmark[1] package is much more precise than `system.time()` with nanosecond rather than millisecond precision. This makes it much easier to compare operations that only take a small amount of time. For example, we can determine the overhead of calling a function: (for an example in the package)

```
library(microbenchmark)

f <- function() NULL
microbenchmark(
  NULL,
  f()
)
#> Unit: nanoseconds
#>   expr min  lq median  uq   max neval
#>   NULL   0  70    139 140   489   100
#>    f() 628 698    768 838 12991   100
```

It's about ~150 ns on my computer (that's the time taken to set up the new environment for the function etc).

[1] http://cran.r-project.org/web/packages/microbenchmark/index.html

It's hard to accurately compute this difference with `system.time` because we need to repeat the operation about a million times, and we get no information about the variability of the estimate. The results may also be systematically biased if some other computation is happening in the background during one of the runs.

```
x <- 1:1e6
system.time(for (i in x) NULL) * 1e3
#>
#>   48    0   49
system.time(for (i in x) f()) * 1e3
#>
#>  244   24  272
```

Running both examples on my computer a few times reveals that the estimate from `system.time` is about 20 nanoseconds higher than the median from `microbenchmark`.

By default, microbenchmark evaluates each expression 100 times, and in random order to control for any systematic variability. It also provides times each expression individually, so you get a distribution of times, which helps estimate error. You can also display the results visually using either `boxplot`, or if you have `ggplot2` loaded, `autoplot`:

```
f <- function() NULL
g <- function() f()
h <- function() g()
i <- function() h()
m <- microbenchmark(
  NULL,
  f(),
  g(),
  h(),
  i())
boxplot(m)

library(ggplot2)
autoplot(m)
```

Microbenchmarking allows you to take the very small parts of a program that profiling has identified as being bottlenecks and explore alternative approaches. It is easier to do this with very small parts of a program because you can rapidly try out alternatives without having to worry too much about correctness (i.e. you are comparing alternatives that are so simple it's obvious whether they're correct or not.)

Useful to think about the first part of the process, generating possible alternatives as brainstorming. You want to come up with as many different approaches to the problem as possible. Don't worry if some of the approaches seem like they will *obviously* be slow: you might be wrong, or that approach might be one step towards a better approach. To get out of a local maxima, you must go down hill.

When doing microbenchmarking, you not only need to figure out what the best method is now, but you need to make sure that fact is recorded somewhere so that when you come back to the code in the future, you remember your reasoning and don't have to redo it again. I find it really useful to write microbenchmarking code as Rmarkdown documents so that I can easily integrate the benchmarking code as well as text describing my hypotheses about why one method is better than another, and listing things that I tried that weren't so effective.

Microbenchmarking is also a powerful tool to improve your intuition about what operations in R are fast and what are slow. The following XXX examples show how to use microbenchmarking to determine the costs of some common R actions, but I really recommend setting up some experiments for the R functions that you use most commonly.

- What's the cost of function vs S3 or S4 method dispatch?
- What's the fastest way to extract a column out of data.frame?

Method dispatch

The following microbenchmark compares the cost of generating one uniform number directly, with a function, with a S3 method, with a S4 method and a R5

```
f <- function(x) NULL

s3 <- function(x) UseMethod("s3")
s3.integer <- function(x) NULL

A <- setClass("A", representation(a = "list"))
setGeneric("s4", function(x) standardGeneric("s4"))
#> [1] "s4"
setMethod(s4, "A", function(x) NULL)
#> [1] "s4"
#> attr(,"package")
#> [1] ".GlobalEnv"

B <- setRefClass("B")
```

```
B$methods(r5 = function(x) NULL)

a <- A()
b <- B$new()

microbenchmark(
  bare = NULL,
  fun = f(),
  s3 = s3(1L),
  s4 = s4(a),
  r5 = b$r5()
)
#> Unit: nanoseconds
#>  expr   min    lq median    uq    max neval
#>  bare    69    70    139   209    489   100
#>   fun   768   908   1047  1257   4051   100
#>    s3  5518  6076   6705  7997  47353   100
#>    s4 19136 20114  21406 23816 150369   100
#>    r5 23746 24654  25666 28879 946559   100
```

On my computer, the bare call takes about 40 ns. Wrapping it in a function adds about an extra 200 ns - this is the cost of creating the environment where the function execution happens. S3 method dispatch adds around 3 µs and S4 around 12 µs.

However, it's important to notice the units: microseconds. There are a million microseconds in a second, so it will take hundreds of thousands of calls before the cost of S3 or S4 dispatch appreciable. Most problems don't involve hundreds of thousands of function calls, so it's unlikely to be a bottleneck in practice.This is why microbenchmarks can not be considered in isolation: they must be carefully considered in the context of your real problem.

Extracting variables out of a data frame

For the plyr package, I did a lot of experimentation to figure out the fastest way of extracting data out of a data frame.

```
n <- 1e5
df <- data.frame(matrix(runif(n * 100), ncol = 100))
x <- df[[1]]
x_ran <- sample(n, 1e3)

microbenchmark(
  x[x_ran],
  df[x_ran, 1],
  df[[1]][x_ran],
  df$X1[x_ran],
  df[["X1"]][x_ran],
  .subset2(df, 1)[x_ran],
  .subset2(df, "X1")[x_ran]
)
#> Unit: microseconds
#>                       expr   min    lq median    uq    max neval
#>                  x[x_ran] 10.76 11.73  12.22 12.85  16.13   100
#>              df[x_ran, 1] 50.01 54.44  63.28 73.54 274.20   100
#>            df[[1]][x_ran] 26.19 28.25  30.91 34.40 201.07   100
#>              df$X1[x_ran] 17.39 19.07  20.46 24.93  51.47   100
#>         df[["X1"]][x_ran] 26.33 27.80  30.56 33.24  52.73   100
#>     .subset2(df, 1)[x_ran] 11.45 12.29  12.92 14.04  18.86   100
#>  .subset2(df, "X1")[x_ran] 11.66 12.47  13.23 14.14  65.02   100
```

Again, the units are in microseconds, so you only need to care if you're doing hundreds of thousands of data frame subsets - but for plyr I am doing that so I do care.

Vectorised operations on a data frame

```
df <- data.frame(a = 1:10, b = -(1:10))
l <- list(0, 10)
l_2 <- list(rep(0, 10), rep(10, 10))
m <- matrix(c(0, 10), ncol = 2, nrow = 10, byrow = TRUE)
df_2 <- as.data.frame(m)
v <- as.numeric(m)

microbenchmark(
  df + v,
  df + l,
  df + l_2,
  df + m,
  df + df_2
)
#> Unit: microseconds
#>       expr   min    lq median    uq    max neval
#>     df + v 610.1 695.8  775.1 877.0 1248.1   100
#>     df + l 409.6 450.1  521.1 606.0 1571.4   100
#>   df + l_2 408.2 453.2  480.1 546.5  785.4   100
#>     df + m 628.9 703.7  813.7 915.2 1869.8   100
#>  df + df_2 449.8 496.4  568.9 653.1 4273.6   100
```

Brainstorming

Most important step is to brainstorm as many possible alternative approaches.

Good to have a variety of approaches to call upon.

- Read blogs
- Algorithm/data structure courses (https://www.coursera.org/course/algs4partI)
- Book
- Read R code

We introduce a few at a high-level in the Rcpp chapter.

Caching

`readRDS`, `saveRDS`, `load`, `save`

Caching packages

Memoisation

A special case of caching is memoisation.

Byte code compilation

R 2.13 introduced a new byte code compiler which can increase the speed of certain types of code 4-5 fold. This improvement is likely to get better in the future as the compiler implements more optimisations - this is an active area of research.

Using the compiler is an easy way to get speed ups - it's easy to use, and if it doesn't work well for your function, then you haven't invested a lot of time in it, and so you haven't lost much.

Other people's code

One of the easiest ways to speed up your code is to find someone who's already done it! Good idea to search for CRAN packages.

```
RppGSL, RcppEigen, RcppArmadillo
```

Stackoverflow can be a useful place to ask.

Important vectorised functions

Not all base functions are fast, but many are. And if you can find the one that best matches your problem you may get big improvements

```
cumsum, diff
rowSums, colSums, rowMeans, colMeans
rle
match
duplicated
```

Read the source code - implementation in C is usually correlated with high performance.

Rewrite in a lower-level language

C, C++ and Fortran are easy. C++ easiest, recommended, and described in the following chapter.

Chapter 17

Profiling and benchmarking

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil" — Donald Knuth.

Your code should be correct, maintainable and fast. Notice that speed comes last - if your function is incorrect or unmaintainable (i.e. will eventually become incorrect) it doesn't matter if it's fast. As computers get faster and R is optimised, your code will get faster all by itself. Your code is never going to automatically become correct or elegant if it is not already.

That said, sometimes there are times where you need to make your code faster: spending several hours of your day might save days of computing time for others. The aim of this chapter is to give you the skills to figure out why your code is slow, what you can do to improve it, and ensure that you don't accidentally make it slow again in the future. You may already be familiar with `system.time`, which tells you how long a block of code takes to run. This is a useful building block, but is a crude tool.

Making fast code is a four part process:

1. Profiling helps you discover parts of your code are taking up the most time
2. Microbenchmarking lets you experiment with small parts of your code to find faster approaches.
3. Timing helps you check that the micro-optimisations have a macro effect, and helps experiment with larger changes (like totally rethinking your approach)
4. A performance testing tool makes sure your code stays fast in the future (e.g. Vbench[1])

[1] http://wesmckinney.com/blog/?p=373

Along the way, you'll also learn about the most common causes of poor performance in R, and how to address them. Sometimes there's no way to improve performance within R, and you'll need to use [[Rcpp]], the topic of the next chapter.

Having a good test suite is important when tuning the performance of your code: you don't want to make your code fast at the expense of making it incorrect. We won't discuss testing any further in this chapter, but we strongly recommend having a good set of test cases written before you begin optimisation.

Good exploration from Winston: http://rpubs.com/wch/3797

Performance profiling

R provides a built in tool for profiling: `Rprof`. When active, this records the current call stack to disk very `interval` seconds. This provides a fine grained report showing how long each function takes. The function `summaryRprof` provides a way to turn this list of call stacks into useful information. But I don't think it's terribly useful, because it makes it hard to see the entire structure of the program at once. Instead, we'll use the `profr` package, which turns the call stack into a data.frame that is easier to manipulate and visualise.

Example showing how to use profr.

Sample pictures.

Timing

Performance testing

Chapter 18

Memory

Understanding how memory works in R can not only help you analyse larger datasets with the same amount of memory, but is also important for writing fast code, as accidental copies are a major cause of slow code. In this chapter, you'll:

- learn how much memory vectors take up by experimenting with `object.size()`
- use the output from `gc()` to explore the net memory impact of a sequence of operations
- understand what garbage collection does, and why you never need to call `gc()` explicitly
- learn how to use the lineprof package to see a line-by-line breakdown of memory used in a bigger script
- explore when R copies an object even though it looks like you're modifying it in place

The chapter will hopefully also help to dispell some myths like:

- You need to call `gc()` regularly to free up more memory.
- For loops in R are always slow.

The details of memory management in R are not documented in one place, but most of the information in this chapter I gleaned from close reading of the documentation (partiularly `?Memory` and `?gc`), the memory profiling[1] section of

[1]http://cran.r-project.org/doc/manuals/R-exts.html#Profiling-R-code-for-memory-use

R-exts, and the SEXPs[2] section of R-ints. The rest I figured out by reading the C source code, performing small experiments and by asking questions on R-devel.

`object.size()`

One of the most useful tools for understanding memory usage in R is `object.size()`, which tells you how much memory an object occupies. This section uses `object.size()` to look at the size of some simple vectors. By exploring some unusual findings, you'll start to understand some important aspects of memory allocation in R.

We'll start with a suprising plot: a line plot of vector length vs. memory size (in bytes) for an integer vector. You might have expected that the size of an empty vector would be 0 and that the memory usage would grow proportionately with length. Neither of those things are true!

```
sizes <- sapply(0:50, function(n) object.size(seq_len(n)))
plot(0:50, sizes, xlab = "Length", ylab = "Bytes", type = "s")
```

This isn't just an artefact of integer vectors: every vector of length 0 occupies 40 bytes of memory:

```
object.size(numeric())
#> 40 bytes
object.size(logical())
#> 40 bytes
object.size(raw())
#> 40 bytes
object.size(list())
#> 40 bytes
```

What are those 40 bytes of memory used for? Every object in R has four components:

- object metadata, the *sxpinfo* (4 bytes). This metadata includes the base type, and information used for debugging and memory management.
- Two pointers: one to the next object in memory, and one to the previous object (2 * 8 bytes). This doubly-linked list makes it easy for internal R code to loop iterate through every object in memory.
- A pointer to the attributes (8 bytes).

[2] http://cran.r-project.org/doc/manuals/R-ints.html#SEXPs

All vector types (e.g. atomic vectors and lists), have three more components:

- The length of the vector (4 bytes). Using 4 bytes should mean that R can only support vectors up to 2 ^ (4 * 8 - 1) (2 ^ 31, about two billion) elements long. But in R 3.0.0 and later you can have vectors up to 2 ^ 52 long: read R-internals to see how support for long vectors[3] was added without changing the size of this field.
- The "true" length of the vector (4 bytes). This is basically never used, except when the object is the hash table for an environment, where the truelength represents the allocated space and the length represents the space currenty used.
- The data (?? bytes). An empty vector has 0 bytes of data, but it's obviously very important otherwise!

If you're counting closely you'll note that only this adds up to 36 bytes. The other 4 bytes are needed as padding after the sxpinfo, so that the pointers start on 8 byte (=64-bit) boundaries. Most process architectures require this alignment for pointers, and even if not required, accessing non-aligned pointers tends to be rather slow.

That explains the intercept on the graph. But why does the memory size grow in irregular jumps? To understand that, you need to know a little bit about how R requests memory from the operating system. Requesting memory, using the `malloc()` function, is a relatively expensive operation, and it would make R slow if it had to request memory every time you created a little vector. Instead, it asks for a big block of memory and then manages it itself: this is called the small vector pool. R uses this pool for vectors less than 128 bytes long, and for efficiency and simplicity, it only allocates vectors that are 8, 16, 32, 48, 64 or 128 bytes long. If we adjust our previous plot by removing the 40 bytes of overhead we can see that those values correspond to the jumps.

```r
plot(0:50, sizes - 40, xlab = "Length", ylab = "Bytes excluding overhead", type = "n")
abline(h = 0, col = "grey80")
abline(h = c(8, 16, 32, 48, 64, 128), col = "grey80")
abline(a = 0, b = 4, col = "grey90", lwd = 4)
lines(sizes - 40, type = "s")
```

It only remains to explain the steps after 128 bytes. While it makes sense for R to manage memory for small vectors, it doesn't make sense to manage it for large vectors: allocating big chunks of memory is something that operating systems are very good at. R always asks for memory in multiples of 8 bytes: this ensures good alignment for the data, in the same way we needed good alignment for the pointers.

[3] http://cran.r-project.org/doc/manuals/R-ints.html#Long-vectors

There are a few other subtleties to `object.size()`: it only promises to give an estimate of the memory usage, not the actual usage. This is because for more complex objects it's not immediately obvious what memory memory usage means. Take environments for example. Using `object.size()` on an environment tells you the size of the environment, not the size of its contents. It would be easy to create a function that did this:

```
env_size <- function(x) {
  if (!is.environment(x)) return(object.size(x))

  objs <- ls(x, all = TRUE)
  sizes <- vapply(objs, function(o) env_size(get(o, x)), double(1))
  structure(sum(sizes) + object.size(x), class = "object_size")
}
object.size(environment())
#> 56 bytes
env_size(environment())
#> 17328 bytes
```

This function isn't quite correct because it's very difficult to cover every special case. For example, you might have an object with an attribute that's an environment that contains a formula which has an environment containing a large object… But even if you could cover all these special cases there's another problem. Environment objects are reference based so you can point to the same object from multiple locations. For example, In the following example, what should the size of `b` be?

```
a <- new.env()
a$x <- 1:1e6

b <- new.env()
b$a <- a

env_size(a)
#> 4000096 bytes
env_size(b)
#> 4000152 bytes
```

You could argue that the size of `b` is actually only 56 bytes, because if you remove `b`, that's how much memory will be freed. But if you deleted `a` first, and then deleted `b` it would free 4000152 bytes. So is the size of `b` 56 or 4000152 bytes? The answer depends on the context.

Another challenge for `object.size()` is strings:

```
object.size("banana")
#> 96 bytes
object.size(rep("banana", 100))
#> 888 bytes
```

On my 64-bit computer, the size of a vector containing "banana" is 96 bytes, but the size of a vector containing 100 "banana"s is 888 bytes. Why the difference? The key is 888 = 96 + 99 * 8. R has a global string pool, which means that every unique string is only stored once in memory. Every other instance of that string is just a pointer, and only needs 8 bytes of storage. `object.size()` does tries to take this into account for individual vectors, but like with environments it's not obvious exactly how the accounting should work.

Exercises

- Repeat the analysis above for numeric, logical, and complex vectors.
- Compare the sizes of the elements of the following two lists. Each contains basically the same data, but one contains vectors of small strings and the other is a single long string.

  ```
  vec <- lapply(0:50, function(i) c("ba", rep("na", i)))
  str <- lapply(vec, paste0, collapse = "")
  ```

- Which takes up more memory: a factor or a character vector? Why?
- Explain the difference in size between `1:5` and `list(1:5)`.

Total memory use

`object.size()` tells you the size of a single object; `gc()` (among other things) tells you the total size of all objects in memory:

```
gc()
#>           used  (Mb) gc trigger (Mb) max used  (Mb)
#> Ncells  649346 34.7    1073225 57.4  1073225  57.4
#> Vcells 2345735 17.9    5397714 41.2 36071175 275.3
```

(we'll get to why it's called `gc()` in the next section)

R breaks down memory usage into Vcells (memory used by vectors) and Ncells (memory used by everything else). But this distinction isn't usually important, and neither are the gc trigger and max used columns. What you're usually most interested in is the total memory used. The function below wraps around `gc()` to return just the amount of memory (in megabytes) that R is currently using.

```
mem <- function() {
  bit <- 8L * .Machine$sizeof.pointer
  if (!(bit == 32L || bit == 64L)) {
    stop("Unknown architecture", call. = FALSE)
  }

  node_size <- if (bit == 32L) 28L else 56L

  usage <- gc()
  sum(usage[, 1] * c(node_size, 8)) / (1024 ^ 2)
}
mem()
#> [1] 52.6
```

Don't expect this number to agree with the amount of memory that your operating system says that R is using:

- Some overhead associated with the R interpreter is not captured by these numbers.
- Both R and the operating system are lazy: they won't try and reclaim memory until it's actually needed. So R might be holding on to memory because the OS hasn't asked for it back yet.
- R counts the memory occupied by objects; there may be gaps from objects that have been deleted. This problem is known as memory fragmentation.

We can build a function of top of `mem()` that tells us how memory changes during the execution of a block of code. We use a little special evaluation to make the code behave in the same way as running it directly. Positive numbers represent an increase in the memory used by R, and negative numbers a decrease.

```
mem_change <- function(code) {
  start <- mem()

  expr <- substitute(code)
  eval(expr, parent.frame())
  rm(code, expr)

  round(mem() - start, 3)
}
# Need about 4 mb to store 1 million integers
mem_change(x <- 1:1e6)
#> [1] 3.815
# We get that memory back when we delete it
mem_change(rm(x))
#> [1] -3.815
```

In the next section, we'll use `mem_change()` to explore how memory is allocated and released by R, and memory is released lazily by the "garbage collector".

Garbarge collection

In some languages you have to explicitly delete unnused objects so that their memory can be returned. R uses an alternative approach, called garbage collection (GC for short), which automatically released memory when an object is no longer used. It does this based on environments and the regular scoping rules: when an environment goes out of scope (for example, when a function finishes executing), all of the contents of that environment are deleted and their memory is freed.

For example, in the following code, a million integers are allocated inside the function, but are automatically removed up when the function terminates. This results in a net change of zero:

```
f <- function() {
  1:1e6
}
mem_change(f())
#> [1] 0
```

This is a little bit of a simplification because in order to find out how much memory is available, our `mem()` function calls `gc()`. As well as returning the amount of memory currently used, `gc()` also triggers garbage collection. Garbage collection normally happens lazily: R calls `gc()` when it needs more space. In reality, that R might hold onto the memory after the function has terminated, but it will release it as soon as it's needed.

Despite what you might have read elsewhere, there's never any point in calling `gc()` yourself, apart to see how much memory is in use. R will automatically run garbage collection whenever it needs more space; if you want to see when that is, call `gcinfo(TRUE)`. The only reason you *might* want to call `gc()` is that it also requests that R should return memory to the operating system. Even that might not have any effect: older versions of Windows had no way for a program to return memory to the OS.

Generally, GC takes care of releasing previously used memory. However, you do need to be aware of situations that can cause memory leaks: when you think you've removed all references to an object, but some are still hanging around so the object never gets freed. In R, the two main causes of memory leaks are formulas and closures. They both capture the enclosing environment, so objects in that environment will not be reclaimed automatically.

The following code illustrates the problem. `f1()` returns the object `10`, so the large vector allocated inside the function will go out of scope and get reclaimed, and the net memory change is 0. `f2()` and `f3()` both return objects that capture environments, and so the net memory change is almost 4 megabytes.

```
f1 <- function() {
  x <- 1:1e6
  10
}
mem_change(x <- f1())
#> [1] 0
x
#> [1] 10
rm(x)

f2 <- function() {
  x <- 1:1e6
  a ~ b
}
mem_change(y <- f2())
#> [1] 3.815
object.size(y)
#> 712 bytes
rm(y)

f3 <- function() {
  x <- 1:1e6
  function() 10
}
mem_change(z <- f3())
#> [1] 3.814
object.size(z)
#> 936 bytes
rm(z)
```

Memory profiling with lineprof

As well as using `mem_change()` to explicitly capture the change in memory caused by running a block of code, we can use memory profiling to automatically capture memory usage every few milliseconds. This functionality is provided by the `utils::Rprof()`, but it doesn't provide a very useful display of the results. Instead, we'll use the lineprof[4] package; it's powered by `Rprof()`, but displays the results in a more informative manner.

[4] https://github.com/hadley/lineprof

To demonstrate lineprof, we're going to explore a minimal implementation of `read.delim` with only three arguments:

```
read_delim <- function(file, header = TRUE, sep = ",") {
  # Determine number of fields by reading first line
  first <- scan(file, what = character(1), nlines = 1, sep = sep, quiet = TRUE)
  p <- length(first)

  # Load all fields as character vectors
  all <- scan(file, what = as.list(rep("character", p)), sep = sep,
    skip = if (header) 1 else 0, quiet = TRUE)

  # Convert from strings to appropriate types (never to factors)
  all[] <- lapply(all, type.convert, as.is = TRUE)

  # Set column names
  if (header) {
    names(all) <- first
  } else {
    names(all) <- paste0("V", seq_along(all))
  }

  # Convert list into data frame
  as.data.frame(all)
}
```

We'll also create a sample csv file to load in:

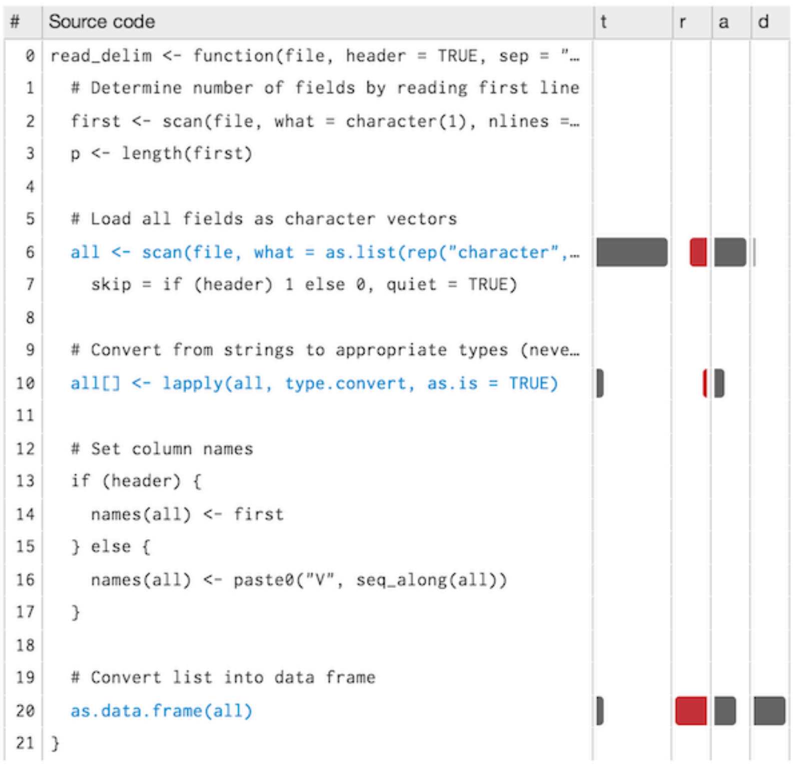
```
library(ggplot2)
write.csv(diamonds, "diamonds.csv", row.names = FALSE)
```

Using lineprof is straightforward. We source the code, then use `lineprof()` with the expression we're interested in, then use `shine()` to view the results. You *must* use `source()` to load the code: you can not create it on the command line. This is because lineprof uses srcrefs to match up the code and run times, and needed srcrefs are only created when you load code from disk.

```
library(lineprof)

source("code/read-delim.R")
prof <- lineprof(read_delim("diamonds.csv"))
shine(prof)
```

`shine()` starts a shiny app which will block your R session. To exit, you'll need to stop the process using escape or ctrl + break. `shine()` will also open a new



```
read_delim <- function(file, header = TRUE, sep = "…
  # Determine number of fields by reading first line
  first <- scan(file, what = character(1), nlines =…
  p <- length(first)

  # Load all fields as character vectors
  all <- scan(file, what = as.list(rep("character",…
    skip = if (header) 1 else 0, quiet = TRUE)

  # Convert from strings to appropriate types (neve…
  all[] <- lapply(all, type.convert, as.is = TRUE)

  # Set column names
  if (header) {
    names(all) <- first
  } else {
    names(all) <- paste0("V", seq_along(all))
  }

  # Convert list into data frame
  as.data.frame(all)
}
```

Figure 18.1: line profiling

web page (or if you're using Rstudio, a new pane) that shows your source code annotated with information about memory usage:

As well as your original source code, there are four columns:

- `t`, the time (in seconds) spent on that line of code
- `a`, the memory (in megabytes) allocated by that line of code.
- `r`, the memory (in megabytes) released by that line of code. While memory allocation is deterministic, memory release is stochastic: it depends on when the GC was run. This means memory release only tells you that the memory release was no longer needed before this line.
- `d`, the number of vector duplications that occured. A vector duplication occurs when R copies a vector to preserve its copy-on-modify semantics.

You can hover over any of the bars to get the exact numbers. For this example, looking at the allocations tells us most of the story:

- `scan()` allocates about 2.5 MB of memory, which is very close to the 2.8 MB of space that the file takes up on disk. You wouldn't expect the numbers to be exactly equal because R doesn't need to store the commas, and the global string pool will save some memory.
- Converting the columns allocates another 0.6 MB of memory. You'd also expect this step to free some memory because we've converted string columns into integer and numeric columns (which occupy less space), but we can't see those releases because GC hasn't been triggered yet.
- Finally, calling `as.data.frame()` on a list allocates about 1.6 megabytes of memory and performs over 600 duplications. This is because `as.data.frame()` isn't terribly efficient and ends up copying the input multiple times. We'll discuss duplications more in the next section.

There are two downsides to profiling:

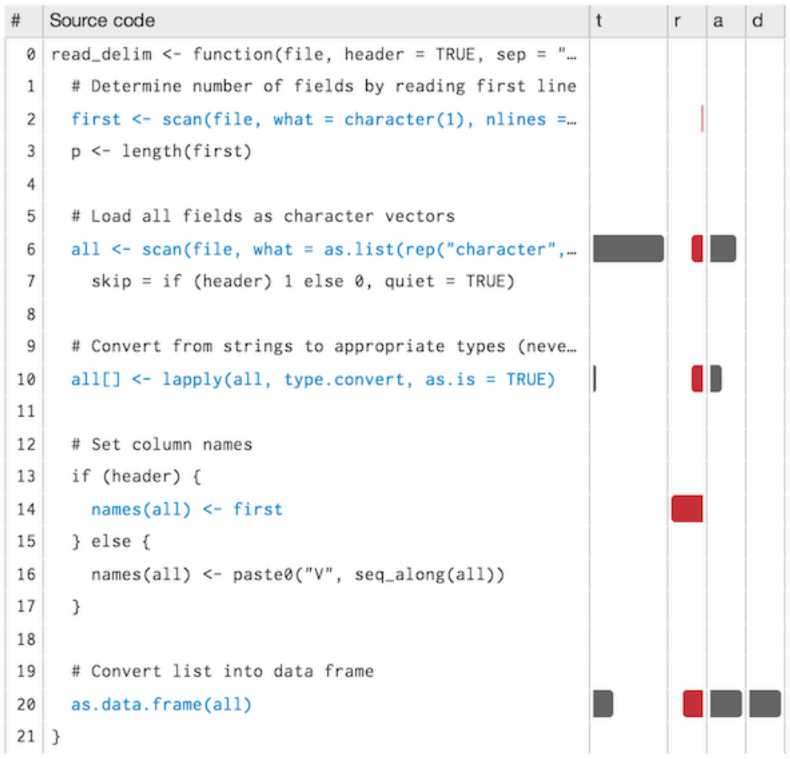
1. `read_delim()` only takes around half a second, and the profile can only capture memory usage at most every 1ms, so we only get about 500 samples.
2. Since GC is lazy, we can never tell exactly when memory is no longer needed.

One way to work around both problems is to use `torture = TRUE`, which forces R to run GC after every allocation (see `gctorture()` for more details). This helps with both problems because memory is freed as soon as possible, and

R runs 10-100x more slowly, so the resolution of the timer is effectively much greater. This allows you to see smaller allocations and exactly when memory is no longer needed.

If we re-run our `read_delim()` example with `torture = TRUE` we get:

```r
prof <- lineprof(read_delim("diamonds.csv"), torture = TRUE)
shine(prof)
```



```
#   Source code                                                  t  r  a  d
0   read_delim <- function(file, header = TRUE, sep = "…
1     # Determine number of fields by reading first line
2     first <- scan(file, what = character(1), nlines =…
3     p <- length(first)
4
5     # Load all fields as character vectors
6     all <- scan(file, what = as.list(rep("character",…
7       skip = if (header) 1 else 0, quiet = TRUE)
8
9     # Convert from strings to appropriate types (neve…
10    all[] <- lapply(all, type.convert, as.is = TRUE)
11
12    # Set column names
13    if (header) {
14      names(all) <- first
15    } else {
16      names(all) <- paste0("V", seq_along(all))
17    }
18
19    # Convert list into data frame
20    as.data.frame(all)
21  }
```

Figure 18.2: line profiling with torture

The basic messages remain the same, but we now we see a big memory release on line 14. Line 14 is the first line after type conversion, so the release represents the memory saved by converting strings to numbers. We still see the large number of duplications with `as.data.frame()` which we'll explore in the next section.

Exercises

- We can make a more efficient `as.data.frame()` when the input is a list by using special knowledge of about the structure of a data frame. A data frame is a list with class `data.frame` and special attribute `row.names`. `row.names` is either a character vector with length matching the columns, or when the row names are sequential integers, the row names are stored in a special format created by `.set_row_names()`. This leads to an alternative `as.data.frame()`:

```
to_df <- function(x) {
  class(x) <- "data.frame"
  attr(x, "row.names") <- .set_row_names(length(x[[1]]))
  x
}
```

What impact does using this function have on `read_delim()`? What are the downsides of this function?

- Line profile the following very simple function with `torture = TRUE`. What is surprising? Read the source code of `gc()` to figure out what's going on.

```
f <- function(n = 1e5) {
  x <- rep(1, n)
  rm(x)
}
```

Modification in place

What happens to `x` in the following code?

```
x <- 1:10
x[5] <- 10
x
#>  [1]  1  2  3  4 10  6  7  8  9 10
```

There's two possibilities:

1. R modifies the existing `x` in place
2. R makes a copy of `x` in a new location, modifies that new vector, and then then changes the name `x` to point to the new location.

It turns out that R can do either depending on the circumstances. In the example above, it will modify in place, but if another variable also points to x, then it will copy it to a new location: To explore what's going on in more detail we need some new tools found in the `pryr` package. Given the name of a variable, `address()` tells us its location in memory, and `refs()` tells us how many names point to that same location.

```
library(pryr)
x <- 1:10
c(address(x), refs(x))
# [1] "0x103100060" "1"

y <- x
c(address(y), refs(y))
# [1] "0x103100060" "2"
```

(Note that if you're using Rstudio this `refs()` will always return two: the environment browser makes a reference to every object you create on the command line, but not inside a function.)

Note that refs is only an estimate and it can only distinguish between 1 and more than 1 references. This means that `refs()` returns 2 in both of the following cases:

```
x <- 1:5
y <- x
rm(y)
# Should really be one, because we've deleted y
refs(x)
#> [1] 2

x <- 1:5
y <- x
z <- x
# Should really be three
refs(x)
#> [1] 2
```

When `refs(x)` is one, modification will occur in place; when `refs(x)` is two, it will make a copy (so that the other pointers to the object contined unchanged). Note that in the following example, `y` keeps pointing to the same location while `x` changes.

```
x <- 1:10
y <- x
c(address(x), address(y))
#> [1] "0x1438d0e8" "0x1438d0e8"

x[5] <- 6L
c(address(x), address(y))
#> [1] "0x41bc368"  "0x1438d0e8"
```

Another useful function is `tracemem()`, which will print a message every time the traced object is copied:

```
x <- 1:10
# Prints the current memory location of the object
tracemem(x)
# [1] "<0x7feeaaa1c6b8>"

x[5] <- 6L

y <- x
# Prints where it has moved from and to
x[5] <- 6L
# tracemem[0x7feeaaa1c6b8 -> 0x7feeaaa1c768]:
```

It's slightly more useful for interactive use than `refs()`, but it's harder to program with (because it just prints a message). (I don't use it very much in this book because it interacts poorly with knitr[5], the tool used to insert the results of R code into the text).

Non-primitive functions that touch the object always increment the ref count. Primitive functions are usually written in such a way that they don't increment the ref count. (The reasons are a little complicated, but see the R-devel thread confused about NAMED[6])

```
x <- 1:10
refs(x)
# [1] 1
mean(x)
refs(x)
# [1] 2

# Touching the object forces an increment
f <- function(x) x
x <- 1:10; f(x); refs(x)
# [1] 2

# Sum is primitive, so doesn't increment
x <- 1:10; sum(x); refs(x)
# [1] 1

# f() and g() never evaluate x so refs doesn't increment
f <- function(x) 10
```

[5]http://yihui.name/knitr/
[6]http://r.789695.n4.nabble.com/Confused-about-NAMED-td4103326.html

```
x <- 1:10; f(x); refs(x)
# [1] 1

g <- function(x) substitute(x)
x <- 1:10; g(x); refs(x)
# [1] 1
```

Generally, any primitive replacement function will modify in place, provided that the object is not referred to elsewhere. This includes `[[<-`, `[<-`, `@<-`, `$<-`, `attr<-`, `attributes<-`, `class<-`, `dim<-`, `dimnames<-`, `names<-`, and `levels<-`. To be precise, all non-primitive functions increment refs, but a primitive function may be written in such a way that it doesn't increment refs. The rules are sufficiently complicated that there's not a lot of point in trying to memorise them; instead approach the problem practically; use `refs()` and `tracemem()` to figure out when objects are being copied.

Once you have determined that where copies are being made, it can be hard to prevent them. If you find yourself resorting to exotic tricks to avoid copies, it may be time to consider switching your function to Rcpp[7].

Loops

For loops in R have a reputation for being slow, but often this slowness is because instead of modifying in place, you're modifying a copy. Take the following code that subtracts the median from each column of a large data.frame:

```
x <- data.frame(matrix(runif(100 * 1e4), ncol = 100))
medians <- vapply(x, median, numeric(1))

system.time({
  for(i in seq_along(medians)) {
    x[, i] <- x[, i] - medians[i]
  }
})
#>
#> 12.04  0.00 12.09
```

It's rather slow - we only have 100 columns and 10,000 rows, but it's taking almost five seconds. We can use `address()` and `refs()` to see what's going on for a small sample of the loop:

```
for(i in 1:5) {
  x[, i] <- x[, i] - medians[i]
```

[7] Rcpp.html

```
  print(c(address(x), refs(x)))
}
#> [1] "0x2d44270" "2"
#> [1] "0x3fecd50" "2"
#> [1] "0x3f2af20" "2"
#> [1] "0x1405a490" "2"
#> [1] "0x1387e890" "2"
```

In each iteration `x` is moved to a new location (copying the complete data frame) and `refs(x)` is always 2. This is because `[<-.data.frame` is not a primitive function, so it always increments the refs. We can make the function substantially more efficient by using either a list or matrix instead of a data frame. Modifying lists and matrices use primitive functions, so the refs are not incremented and all modifications are in place.

```
y <- as.list(x)
system.time({
  for(i in seq_along(medians)) {
    y[[i]] <- y[[i]] - medians[i]
  }
})
#>
#> 0.012 0.000 0.012

z <- as.matrix(x)
system.time({
  for(i in seq_along(medians)) {
    z[, i] <- z[, i] - medians[i]
  }
})
#>
#> 0.044 0.000 0.043
```

Exercises

- The code below makes one duplication. Where does it occur and why? (Hint: look at `refs(y)`)

  ```
  y <- as.list(x)
  for(i in seq_along(medians)) {
    y[[i]] <- y[[i]] - medians[i]
  }
  ```

- The implementation of `as.data.frame()` in the previous section has one big downside. What is it and how could you avoid it?

Chapter 19

High performance functions with Rcpp

Sometimes R code just isn't fast enough - you've used profiling to find the bottleneck, but there's simply no way to make the code any faster. This chapter is the answer to that problem. You'll learn how to rewrite key functions in C++ to get much better performance, while not taking too much longer to write. The key to this magic is Rcpp[1], a fantastic tool written by Dirk Eddelbuettel and Romain Francois (with key contributions by Doug Bates, John Chambers and JJ Allaire), that makes it dead simple to connect C++ to R. It is *possible* to write C or Fortran code for use in R, but it will be painful; Rcpp provides a clean, approachable API that lets you write high-performance code, insulated from R's arcane C API.

Typical bottlenecks that C++ can help with are:

- Loops that can't easily be vectorised because each iteration depends on the previous. C++ modifies objects in place, so there is little overhead when modifying a data structure many times.
- Recursive functions, or problems which involve calling functions millions of times. The overhead of calling a function in C++ is much lower than the overhead of calling a function in R. To give you some idea of the magnitude, on my computer when writing this book the overhead in C++ was ~5ns compared to ~200ns for R.
- Problems that require advanced data structures and algorithms that R doesn't provide. Through the standard template library (STL), C++ has efficient implementations of many important data structures, from ordered maps to double ended queues.

[1]http://dirk.eddelbuettel.com/code/rcpp.html

Rewriting a function in C++ can lead to a 2-3 order of magnitude speed up, but most improvements will be more modest. While pure R code is relatively slow compared to C or C++, many bottlenecks in base R have already been replaced with hand-written C functions. This means that if your function already uses vectorised operations, you are unlikely to see a large improvement in performance. Note, however, that if you do rewrite a base C function with Rcpp, it's likely to be much shorter, because you can use the more sophisticated tools provided by C++.

The aim of this chapter is to give you the absolute necessities of C++ and Rcpp. A working knowledge of C++ is helpful, but not essential. Many good tutorials and references are freely available, including [http://www.learncpp.com/] and [http://www.cplusplus.com/]. For more advanced topics, the "Effective C++" series by Scott Meyers is popular choice. Dirk Eddelbuettel has written an entire book on Rcpp, "Seamless R and C++ integration with Rcpp"[2], which provides much more detail than we can here. If you're serious about Rcpp, make sure to get that book!

In this chapter you'll learn:

- How to write C++ code by seeing R functions and their C++ equivalents.
- Important Rcpp classes and methods
- How to use Rcpp "sugar" to avoid C++ loops and convert directly from vectorised R code
- How to work with missing values
- Some of the most important techniques, data structures and algorithms from standard template library (STL)

The chapter concludes with a selection of real case studies showing how others have used C++ and Rcpp to speed up their slow R code.

Getting started

All examples in this chapter need at least version 0.10.1 of the `Rcpp` package. This version includes `cppFunction` and `sourceCpp`, which makes it very easy to connect C++ to R. You'll also (obviously!) need a working C++ compiler.

`cppFunction` allows you to write C++ functions in R like this:

```
library(Rcpp)
cppFunction('
  int add(int x, int y, int z) {
    int sum = x + y + z;
```

[2] http://www.springer.com/statistics/computational+statistics/book/978-1-4614-6867-7

```
    return sum;
  }'
)
add # like a regular R function, printing displays info about the function
#> function (x, y, z)
#> .Primitive(".Call")(<pointer: 0x7fa688ff3590>, x, y, z)
#> <environment: 0x37db310>
add(1, 2, 3)
#> [1] 6
```

When you run this code, Rcpp will compile the C++ code and construct an R function that connects to the compiled C++ function. If you're familiar with `inline::cxxfunction`, `cppFunction` is similar, except that you specify the function completely in the string, and it parses the C++ function arguments to figure out what the R function arguments should be.

Getting started with C++

C++ is a large language, and there's no way to cover it exhaustively here. Our aim here is to give you the basics so you can start writing useful functions that allow you to speed up slow parts of your R code. We'll spend minimal time on advanced features like object oriented programming and templates, because our focus is not on writing big programs in C++, just small, self-contained function.

The following section shows the basics of C++ by translating simple R functions to their C++ equivalents. We'll start simple with a function that has no inputs and a scalar output, and then get progressively more complicated:

- Scalar input and scalar output
- Vector input and scalar output
- Vector input and vector output
- Matrix input and vector output

No inputs, scalar output

Let's start with a very simple function. It has no arguments and always returns the integer 1:

```
one <- function() 1L
```

The equivalent C++ function is:

```
int one() {
  return 1;
}
```

We can compile and use this from R with `cppFunction`

```
cppFunction('
  int one() {
    return 1;
  }
')
```

This small function illustrates a number of important differences between R and C++:

- The syntax to create a function looks like the syntax to call a function; you don't use assignment to create functions like in R.
- You must declare the type of output the function returns. This function returns an `int` (a scalar integer). The classes for the most common types of R vectors are: `NumericVector`, `IntegerVector`, `CharacterVector` and `LogicalVector`.
- Scalars and vectors are different. The scalar equivalents of numeric, integer, character and logical vectors are: `double`, `int`, `String` and `bool`.
- You must use an explicit `return` statement to return a value from the function.
- Every statement is terminated by a `;`.

Scalar input, scalar output

The next example function makes things a little more complicated by implementating a scalar version of the `sign` function which returns 1 if the input is positive, and -1 if it's negative:

```
signR <- function(x) {
  if (x > 0) {
    1
  } else if (x == 0) {
    0
  } else {
    -1
  }
}

cppFunction('
  int signC(int x) {
    if (x > 0) {
      return 1;
    } else if (x == 0) {
      return 0;
    } else {
      return -1;
    }
  }'
)
```

In the C++ version:

- We declare the type of each input in the same way we declare the type of the output. While this makes the code a little more verbose, it also makes it very obvious what type of input the function needs. Similarly to S3 and S4 in R, C++ allows you to write different functions with the same name that have different inputs (number or type).
- The `if` syntax is identical - while there are some big differences between R and C++, there are also lots of similarities! C++ also has a `while` statement that works the same way as R's. You can also use `break`, but use `continue` instead of `next`.

Vector input, scalar output

One big difference between R and C++ is that the cost of loops is much lower. For example, we could implement the `sum` function in R using a loop. If you've been programming in R a while, you'll probably have a visceral reaction to this function!

```
sumR <- function(x) {
  total <- 0
  for (i in seq_along(x)) {
    total <- total + x[i]
  }
  total
}
```

In C++, loops have very little overhead, so it's fine to use them (later, we'll see alternatives to `for` loops that more clearly express your intent; they're not faster, but they can make your code easier to understand).

```
cppFunction('
  double sumC(NumericVector x) {
    int n = x.size();
    double total = 0;
    for(int i = 0; i < n; ++i) {
      total += x[i];
    }
    return total;
  }
')
```

The C++ version is similar, but:

- To find the length of the vector, we use the `size()` method, which returns an integer. Again, whenever we create a new variable we have to tell C++ what type of object it will hold. An `int` is a scalar integer, but we could have used `double` for a scalar numeric, `bool` for a scalar logical, or a `String` for a scalar string.
- The `for` statement has a different syntax: `for(intialisation; condition; increase)`. The initialise component creates a new variable called `i` and sets it equal to 0. The condition is checked in each iteration of the loop: the loop is continues while it's `true`. The increase statement is run after each loop iteration, but before the condition is checked. Here we use the special prefix operator `++` which increases the value of `i` by 1.
- Vectors in C++ start at 0. I'll say this again because it's so important: VECTORS IN C++ START AT 0! This is a very common source of bugs when converting R functions to C++.
- We can't use `<-` (or `->`) for assignment, but instead use `=`.
- We can take advantage of the in-place modification operators: `total += x[i]` is equivalent to `total = total + x[i]`. Similar in-place operators are `-=`, `*=` and `/=`. This is known as a side-effect, where the function `++` or `+=` modifies its argument `i` or `x` without us asking. Functions in R rarely have side-effects, and we need to be careful that our Rcpp functions don't modify their inputs. More on that later.

This is a good example of where C++ is much more efficient than the R equivalent. As shown by the following microbenchmark, our `sumC` function is competitive with the built-in (and highly optimised) `sum` function, while `sumR` is several orders of magnitude slower.

```
library(microbenchmark)
x <- runif(1e3)
```

```
microbenchmark(
  sum(x),
  sumR(x),
  sumC(x)
)
#> Unit: microseconds
#>     expr      min       lq   median       uq     max neval
#>   sum(x)    2.514    2.724    3.771    7.054   18.51   100
#>  sumR(x) 1133.805 1166.768 1214.226 1248.412 3834.78   100
#>  sumC(x)    5.028    6.216    7.997   18.543   43.93   100
```

Vector input, vector output

For our next example, we'll create a function that computes the distance between one value and a vector of other values:

```
pdistR <- function(x, ys) {
  sqrt( (x - ys) ^ 2 )
}
```

From the function definition, it's not obvious that we want `x` to be a scalar - that's something we'd need to mention in the documentation. That's not a problem in the C++ version:

```
cppFunction('
  NumericVector pdistC(double x, NumericVector ys) {
    int n = ys.size();
    NumericVector out(n);

    for(int i = 0; i < n; ++i) {
      out[i] = sqrt(pow(ys[i] - x, 2.0));
    }
    return out;
  }
')
```

This function introduces only a few new concepts:

- We create a new numeric vector of length `n` with a constructor: `NumericVector out(n)`. Another useful way of making a vector is to copy an existing vector: `NumericVector zs = clone(ys)`.
- C++ doesn't use `^` for exponentiation, it instead uses the `pow` function.

Note that because the R function is fully vectorised, it is already going to be fast. On my computer, it takes around 8 ms with a 1 million element `y` vector. The C++ function is twice as fast, ~4 ms, but assuming it took you 10 minutes to write the C++ function, you'd need to run it ~150,000 times to make it a net saver of time. The reason why the C++ function is faster is subtle, and relates to memory management. The R version needs to create an intermediate vector the same length as y (`x - ys`), and allocating memory is an expensive operation. The C++ function avoids this overhead because it uses an intermediate scalar.

In the sugar section, you'll see how to rewrite this function to take advantage of Rcpp's vectorised operations so that the C++ code is barely longer than the R code.

Matrix input, vector output

Each vector type also has a matrix equivalent: `NumericMatrix`, `IntegerMatrix`, `CharacterMatrix` and `LogicalMatrix`. Using them is straightforward. For example, we could create a function that reproduces `rowSums`:

```
cppFunction('
  NumericVector rowSumsC(NumericMatrix x) {
    int nrow = x.nrow(), ncol = x.ncol();
    NumericVector out(nrow);

    for (int i = 0; i < nrow; i++) {
      double total = 0;
      for (int j = 0; j < ncol; j++) {
        total += x(i, j);
      }
      out[i] = total;
    }
    return out;
  }
')
x <- matrix(sample(100), 10)
rowSums(x)
#>  [1] 482 450 440 284 482 664 603 544 475 626
rowSumsC(x)
#>  [1] 482 450 440 284 482 664 603 544 475 626
```

The main thing to notice is that when subsetting a matrix we use `()` and not `[]`, and that matrix objects have `nrow()` and `ncol()` methods.

Using sourceCpp

To simplify the initial presentation the examples in this section have used inline C++ via `cppFunction`. For real problems, it's usually easier to use standalone C++ files and then source them into R using the `sourceCpp` function. This will enable you to take advantage of text editor support for C++ files (e.g. syntax highlighting) as well as make it easier to identify the line numbers of compilation errors. Standalone C++ files can also contain embedded R code in special C++ comment blocks. This is really convenient if you want to run some R test code.

Your standalone C++ file should have extension `.cpp`, and needs to start with:

```
#include <Rcpp.h>
using namespace Rcpp;
```

And for each function that you want available within R, you need to prefix it with:

```
// [[Rcpp::export]]
```

(Note that the space is mandatory)

(This is somewhat similar to roxygen2's `@export` tag, but `Rcpp::export` controls whether a function is exported from C++ to R, where `@export` controls whether a function is exported from a package and made available to a package user.)

Then using `sourceCpp("path/to/file.cpp")` will compile the C++ code, create the matching R functions and add them to your current session. (Note that these functions will not persist across `save()` and `load()`, such as when you restore your workspace.)

For example, running `sourceCpp` on the following file first compiles the C++ code and then compares it to native equivalent:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double meanC(NumericVector x) {
  int n = x.size();
  double total = 0;

  for(int i = 0; i < n; ++i) {
    total += x[i] / n;
  }
  return total;
}

/*** R
  library(microbenchmark)
  x <- runif(1e5)
  microbenchmark(
    mean(x),
    meanC(x))
*/
```

The R code is run with `source(echo = TRUE)` so you don't need to explicitly print output.

For the remainder of this chapter C++ code will typically be presented standalone rather than wrapped in a call to `cppFunction`. If you want to try compiling and/or modifying the examples you should paste them into a C++ source file that includes the elements described above.

Exercises

With the basics of C++ in hand, now is a great time to practice by reading and writing some simple C++ functions.

For each of the following C++ functions, read the code and figure out what base R function it corresponds to. You might not understand every part of the code yet, but you should be able to figure out the basics of what the function does.

```
double f1(NumericVector x) {
  int n = x.size();
  double y = 0;

  for(int i = 0; i < n; ++i) {
    y += x[i] / n;
  }
  return y;
}

NumericVector f2(NumericVector x) {
  int n = x.size();
  NumericVector out(n);

  out[0] = x[0];
  for(int i = 1; i < n; ++i) {
    out[i] = out[i - 1] + x[i];
  }
  return out;
}

bool f3(LogicalVector x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    if (x[i]) return true;
  }
  return false;
}

int f4(Function pred, List x) {
  int n = x.size();

  for(int i = 0; i < n; ++i) {
    LogicalVector res = pred(x[i]);
    if (res[0]) return i + 1;
  }
  return 0;
}

NumericVector f5(NumericVector x, NumericVector y) {
  int n = std::max(x.size(), y.size());
  NumericVector x1 = rep_len(x, n);
  NumericVector y1 = rep_len(y, n);

  NumericVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = std::min(x1[i], y1[i]);
  }

  return out;
}
```

To practice your function writing skills, convert the following functions into C++. For now, assume the inputs have no missing values.

- `all`
- `cumprod`, `cummin`, `cummax`.
- `diff`. Start by assuming lag 1, and then generalise for lag n.

- `range`.
- `var`. Read about the approaches you can take at wikipedia[3]. Whenever implementing a numerical algorithm it's always good to check what is already known about the problem.

Rcpp classes and methods

You've already seen the basic vector classes (`IntegerVector`, `NumericVector`, `LogicalVector`, `CharacterVector`) and their scalar (`int`, `double`, `bool`, `String`) and matrix (`IntegerMatrix`, `NumericMatrix`, `LogicalMatrix`, `CharacterMatrix`) equivalents.

All R objects have attributes, which can be queried and modified with the `attr` method. Rcpp also provides a `names()` method for the commonly used attribute: `attr("names")`. The following code snippet illustrates these methods. Note the use of the `create()` class method to easily create an R vector from C++ scalar values.

```
// [[Rcpp::export]]
NumericVector attribs() {
  NumericVector out = NumericVector::create(1, 2, 3);

  out.names() = CharacterVector::create("a", "b", "c");
  out.attr("my-attr") = "my-value";
  out.attr("class") = "my-class";

  return out;
}
```

You can use the `slot()` method in a similar way to get and set slots of S4 objects.

Rcpp also provides classes `List` and `DataFrame`. These are more useful for output than input, because lists and data frames can contain arbitrary classes, and this does not fit well with C++'s desire to have the types of all inputs known in advance. If, however, the list is an S3 object with components of known types, you can extract the components and manually convert to their C++ equivalents with `as`.

For example, the linear model objects that `lm` produces are lists and the components are always of the same type. The following code illustrates how you might component the mean percentage error (`mpe`) of a linear model. This isn't a very good example for when you might use C++ (because it's so easily implemented

[3]http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

in R), but it illustrates how to pull out the components of a list. Note the use of the `inherits()` method and the `stop()` function to check that the object really is a linear model.

```
// [[Rcpp::export]]
double mpe(List mod) {
  if (!mod.inherits("lm")) stop("Input must be a linear model");

  NumericVector resid = as<NumericVector>(mod["residuals"]);
  NumericVector fitted = as<NumericVector>(mod["fitted.values"]);

  int n = resid.size();
  double err = 0;
  for(int i = 0; i < n; ++i) {
    err += resid[i] / (fitted[i] + resid[i]);
  }
  return err / n;
}

/*** R
  mod <- lm(mpg ~ wt, data = mtcars)
  mpe(mod)
*/
```

It is possible to write code that works differently depending on the type of the R input, but it is beyond the scope of this book.

You can put R functions in an object of type `Function`. Calling an R function from C++ is straightforward. The string constructor of the function object will look for a function of that name in the global environment.

```
Function assign("assign");
```

You can call functions with arguments specified by position:

```
assign("y", 1);
```

Or by name, with a special syntax:

```
assign(_["x"] = "y", _["value"] = 1);
```

The challenge is storing the output. If you don't know in advance what the output will be, store it in an `RObject` or in components of a `List`. For example, the following code is a basic implementation of `lapply` in C++:

```
// [[Rcpp::export]]
List lapply1(List input, Function f) {
  int n = input.size();
  List out(n);

  for(int i = 0; i < n; i++) {
    out[i] = f(input[i]);
  }

  return out;
}

/*** R
  lapply1(1:10, sqrt)
  lapply1(list("a", 1, F), class)
*/
```

There are also classes for many more specialised language objects: `Environment`, `ComplexVector`, `RawVector`, `DottedPair`, `Language`, `Promise`, `Symbol`, `WeakReference` and so on. These are beyond the scope of this chapter and won't be discussed further.

Rcpp sugar

Rcpp provides a lot of "sugar", C++ functions that work very similarly to their R equivalents. (The main difference is that they don't recycle their inputs - you need to do that yourself). Rcpp sugar makes it possible to write efficient C++ code that looks almost identical to the R equivalent. If a sugar version of the function you're interested exists, you should use it: it's expressive and well tested. Sugar functions aren't always faster than your hand-written equivalent, but they will get faster as more time is spent on optimising Rcpp.

Sugar functions can be roughly broken down into

- arithmetic and logical operators
- logical summary functions
- vector views
- other useful functions

Arithmetic and logical operators

All the basic arithmetic and logical operators are vectorised: `+ *`, `-`, `/`, `pow`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `!`. For example, we could use sugar to considerably simplify the implementation of our `pdistC` function. (If you don't remember I've included the R version of `pdistC`, `pdistR`, as well. Note the similarities with the Rcpp sugar version.)

```
pdistR <- function(x, ys) {
  (x - ys) ^ 2
}

// [[Rcpp::export]]
NumericVector pdistC2(double x, NumericVector ys) {
  return pow((x - ys), 2);
}
```

Logical summary functions

The sugar function `any` and `all` are fully lazy, so that e.g `any(x == 0)` might only need to evaluate one element of the value, and return a special type that can be converted into a `bool` using `is_true`, `is_false`, or `is_na`.

For example, we could use this to write an efficient funtion to determine whether or not a numeric vector contains any missing values. In R we could do `any(is.na(x))`:

```
any_naR <- function(x) any(is.na(x))
```

However that will do almost the same amount of work whether there's a missing value in the first position or the last. Here's the C++ implementation:

```
// [[Rcpp::export]]
bool any_naC(NumericVector x) {
  return is_true(any(is_na(x)));
}
```

Our C++ `any_naC` function is slightly slower than `any_naR` when there are no missing values, or the missing value is at the end, but it's much faster when the first value is missing.

```
library(microbenchmark)
x0 <- runif(1e5)
x1 <- c(x0, NA)
x2 <- c(NA, x0)

microbenchmark(
  any_naR(x0), any_naC(x0),
  any_naR(x1), any_naC(x1),
  any_naR(x2), any_naC(x2))
```

Vector views

A number of helpful functions provide a "view" of a vector: `head`, `tail`, `rep_each`, `rep_len`, `rev`, `seq_along`, `seq_len`. In R these would all produce copies of the vector, but in Rcpp they simply point to the existing vector and override the subsetting operator (`[`) to implement special behaviour. This makes them very efficient: `rep_len(x, 1e6)` does not have to make a million copies of x.

Other useful functions

Finally, there are a grab bag of sugar functions that mimic frequently used R functions:

- Math functions: `abs`, `acos`, `asin`, `atan`, `beta`, `ceil`, `ceiling`, `choose`, `cos`, `cosh`, `digamma`, `exp`, `expm1`, `factorial`, `floor`, `gamma`, `lbeta`, `lchoose`, `lfactorial`, `lgamma`, `log`, `log10`, `log1p`, `pentagamma`, `psigamma`, `round`, `signif`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `tetragamma`, `trigamma`, `trunc`,
- Scalar summaries: `mean`, `min`, `max`, `range`, `sum`, `sd` and `var`.
- Vector summaries: `cumsum`, `diff`, `pmin`, and `pmax`
- Finding values: `match`, `self_match`, `which_max`, `which_min`
- `duplicated`, `unique`
- `d/q/p/r` for all standard distributions in R.
- `noNA(x)`: this asserts that the vector `x` does not contain any missing values, and allows optimisation of some mathematical operations.

Missing values

If you're working with missing values, you need to know two things:

- how R's missing values behave in C++'s scalars (e.g. `double`)
- how to get and set missing values in vectors (e.g. `NumericVector`)

Scalars

The following code explores what happens when you take one of R's missing values, coerce it into a scalar, and then coerce back to an R vector. This is a generally useful technique: if you don't know what an operation will do, design an experiment to figure it out.

```
// [[Rcpp::export]]
List scalar_missings() {
  int int_s = NA_INTEGER;
  String chr_s = NA_STRING;
  bool lgl_s = NA_LOGICAL;
  double num_s = NA_REAL;

  return List::create(int_s, chr_s, lgl_s, num_s);
}

/*** R
  str(scalar_missings())
*/
```

Things look pretty good here: with the exception of `bool`, all of the missing values have been preserved.

Integers

However, things are not quite as they seem for integers. Missing values are stored as the smallest integer so stored so if you don't do anything to them, they'll be preserved, but C++ doesn't know that the smallest integer has special behaviour so if you do anything with it you're likely to get an incorrect value: `evalCpp('NA_INTEGER + 1')` gives -2147483647.

If you want to work with missing values in integers, either use length one `IntegerVectors` or be very careful with your code.

Doubles

If you're working with doubles, you may be able to get away with ignoring missing values and working with NaN (not a number). R's missing values are a special type of the IEEE 754 floating point number NaN. That means if you coerce them to `double` in your C++ code, they will behave like regular NaN's. That means, in a logical context they always evaluate to FALSE:

```
evalCpp("NAN == 1")
evalCpp("NAN < 1")
evalCpp("NAN > 1")
evalCpp("NAN == NAN")
```

But be careful when combining then with boolean values:

```
evalCpp("NAN && TRUE")
evalCpp("NAN || FALSE")
```

In numeric contexts, they propagate similarly to NA in R:

```
evalCpp("NAN + 1")
evalCpp("NAN - 1")
evalCpp("NAN / 1")
evalCpp("NAN * 1")
```

Strings

`String` is an scalar string class introduced by Rcpp, so it knows how to deal with missing values.

Boolean

C++'s `bool` only stores two values (true or false), but R's logical vector has three possible values (TRUE, FALSE and NA). If you coerce a length 1 logical vector, first make sure it doesn't contain any missing values otherwise they will be converted to TRUE.

Vectors

To set a missing value in a vector, you need to use a missing value specific to the type of vector. Unfortunately these are not named terribly consistently:

```
// [[Rcpp::export]]
List missing_sampler() {
  return(List::create(
    NumericVector::create(NA_REAL),
    IntegerVector::create(NA_INTEGER),
    LogicalVector::create(NA_LOGICAL),
    CharacterVector::create(NA_STRING)));
}

/*** R
  str(missing_sampler())
*/
```

To check if a value in a vector is missing, use the class method `is_na`:

```
// [[Rcpp::export]]
LogicalVector is_naC(NumericVector x) {
  int n = x.size();
  LogicalVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = NumericVector::is_na(x[i]);
  }
  return out;
}

/*** R
  is_naC(c(NA, 5.4, 3.2, NA))
*/
```

Another alternative is the similarly named sugar function `is_na`: it takes a vector and returns a logical vector.

```
// [[Rcpp::export]]
LogicalVector is_naC2(NumericVector x) {
  return is_na(x);
}

/*** R
is_naC2(c(NA, 5.4, 3.2, NA))
*/
```

Exercises

- Rewrite any of the functions from the first exercises to correctly deal with missing values. If `na.rm` is true, ignore the missing values. If `na.rm` is false, return missing values the first time a missing value is encountered. Some functions you can practice with are: `min`, `max`, `range`, `mean`, `var`
- `cumsum` and `diff` need slightly more complicated behaviour for missing values.

The STL

The real strength of C++ shows itself when you need to implement more complex algorithms. The standard template library (STL) provides set of extremely useful data structures and algorithms. This section will explain the most important algorithms and data structures and point you in the right direction to learn more. We can't teach you everything you need to know about the STL, but hopefully the examples will at least show you the power of the STL, and persuade that it's useful to learn more.

If you need an algorithm or data structure that isn't implemented in STL, a good place to look is boost[4]. Installing boost on to your computer is beyond the scope of this chapter, but once you have it installed, you can use `boost` data structures and algorithms by including the appropriate header file with (e.g.) `#include <boost/array.hpp>`.

Using iterators

Iterators are used extensively in the STL: many functions either accept or return iterators. They are the next step up from basic loops, abstracting away the details of the underlying data structure. Iterators have three main operators: they can be advanced with `++`, dereferenced (to get the value they refer to) with `*` and compared using `==`. For example we could re-write our sum function using iterators:

```
// [[Rcpp::export]]
double sum3(NumericVector x) {
  double total = 0;

  for(NumericVector::iterator it = x.begin(); it != x.end(); ++it) {
    total += *it;
  }
  return total;
}
```

The main changes are in the for loop:

- We start at `x.begin()` and loop until we get to `x.end()`. A small optimization is to store the value of the end iterator so we don't need to look it up each time. This only saves about 2 ns per iteration, so it's only important when the calculations in the loop are very simple.
- Instead of indexing into x, we use the dereference operator to get its current value: `*it`.
- Notice the type of the iterator: `NumericVector::iterator`. Each vector type has its own iterator type: `LogicalVector::iterator`, `CharacterVector::iterator` etc.

[4]http://www.boost.org/doc/

Iterators also allow us to use the C++ equivalents of the apply family of functions. For example, we could again rewrite `sum` to use the `accumulate` function, which takes an starting and ending iterator and adds all the values in between. The third argument to accumulate gives the initial value: it's particularly important because this also determines the data type that accumulate uses (here we use `0.0` and not `0` so that accumulate uses a `double`, not an `int`.). To use `accumulate` we need to include the `<numeric>` header.

```cpp
#include <numeric>

// [[Rcpp::export]]
double sum4(NumericVector x) {
  return std::accumulate(x.begin(), x.end(), 0.0);
}
```

`accumulate` (along with the other functions in `<numeric>`, `adjacent_difference`, `inner_product` and `partial_sum`) are not that important in Rcpp because Rcpp sugar provides equivalents.

Algorithms

The `<algorithm>` header provides a large number of algorithms that work with iterators. For example, we could write a basic Rcpp version of `findInterval` that takes two arguments a vector of values and a vector of breaks - the aim is to find the bin that each x falls into. This shows off a few more advanced iterator features. Read the code below and see if you can figure out how it works.

```cpp
#include <algorithm>

// [[Rcpp::export]]
IntegerVector findInterval2(NumericVector x, NumericVector breaks) {
  IntegerVector out(x.size());

  NumericVector::iterator it, pos;
  IntegerVector::iterator out_it;

  for(it = x.begin(), out_it = out.begin(); it != x.end(); ++it, ++out_it) {
    pos = std::upper_bound(breaks.begin(), breaks.end(), *it);
    *out_it = std::distance(breaks.begin(), pos);
  }

  return out;
}
```

The key points are:

- We step through two iterators (input and output) simultaneously.
- We can assign into an dereferenced iterator (`out_it`) to change the values in `out`.
- `upper_bound` returns an iterator. If we wanted the value of the `upper_bound` we could dereference it; to figure out its location, we use the `distance` function.
- Small note: if we want this function to be as fast as `findInterval` in R (which uses hand-written C code), we need to compute the calls to `.begin()` and `.end()` once and save the results. This is easy, but it distracts from this example so it has been omitted. Making this change yields a function that's slightly faster than R's `findInterval` function, but is about 1/10 of the code.

It's generally better to use algorithms from the STL than hand rolled loops. In "Effective STL", Scott Meyer gives three reasons: efficiency, correctness and maintainability. Algorithms from the STL are written by C++ experts to be extremely efficient, and they have been around for a long time so they are well tested. Using standard algorithms also makes the intent of your code more clear, helping to make it more readable and more maintainable.

A good reference guide for algorithms is http://www.cplusplus.com/reference/algorithm/

Data structures

The STL provides a large set of data structures: `array`, `bitset`, `list`, `forward_list`, `map`, `multimap`, `multiset`, `priority_queue`, `queue`, `dequeue`, `set`, `stack`, `unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`, and `vector`. The most important of these datastructures are the `vector`, the `unordered_set`, and the `unordered_map`. We'll focus on these three in this section, but using the others is similar: they just have different performance tradeoffs. For example, the `deque` (pronounced "deck") has a very similar interface to vectors but a different underlying implementation that has different performance trade-offs. You may want to try them for your problem. A good reference for STL data structures is http://www.cplusplus.com/reference/stl/ - I recommend you keep it open while working with the STL.

Rcpp knows how to convert from many STL data structures to their R equivalents, so you can return them from your functions without explicitly converting to R data structures.

Vectors

An STL vector is very similar to an R vector, except that it expands efficiently. This makes vectors appropriate to use when you don't know in advance how big the output will be. Vectors are templated, which means that you need to specify the type of object the vector will contain when you create it: `vector<int>`, `vector<bool>`, `vector<double>`, `vector<String>`. You can access individual elements of a vector using the standard `[]` notation, and you can add a new element to the end of the vector using `.push_back()`. If you have some idea in advance how big the vector will be, you can use `.reserve()` to allocate sufficient storage.

The following code implements run length encoding (`rle`). It produces two vectors of output: a vector of values, and a vector `lengths` giving how many times each element is repeated. It works by looping through the input vector `x` comparing each value to the previous: if it's the same, then it increments the last value in `lengths`; if it's different, it adds the value to the end of `values`, and sets the corresponding length to 1.

```
// [[Rcpp::export]]
List rleC(NumericVector x) {
  std::vector<int> lengths;
  std::vector<double> values;

  // Initialise first value
  int i = 0;
  double prev = x[0];
  values.push_back(prev);
  lengths.push_back(1);

  for(NumericVector::iterator it = x.begin() + 1; it != x.end(); ++it) {
    if (prev == *it) {
      lengths[i]++;
    } else {
      values.push_back(*it);
      lengths.push_back(1);

      i++;
      prev = *it;
    }
  }

  return List::create(_["lengths"] = lengths, _["values"] = values);
}
```

(An alternative implementation would be to replace `i` with the iterator `lengths.rbegin()` which always points to the last element of the vector - you might want to try implementing that yourself.)

Other methods of a vector are described at http://www.cplusplus.com/reference/vector/vector/.

Sets

Sets maintain a unique set of values, and can efficiently tell if you've seen a value before. They are useful for problems that involve duplicates or unique values (like `unique`, `duplicated`, or `in`). C++ provides both ordered (`std::set`) and unordered sets (`std::tr1::unorded_set`), depending on whether or not order matters for you. Unordered sets tend to be much faster (because they use a hash table internally rather than a tree), so even if you need an ordered set, you should consider using an unordered set and then sorting the output. Like vectors, sets are templated, so you need to request the appropriate type of set for your purpose: `unordered_set<int>`, `unordered_set<bool>`, etc.

http://www.cplusplus.com/reference/set/set/ and http://www.cplusplus.com/reference/unordered_set/ur
provide complete documentation for sets structures.

The following function uses an unordered set to implement an equivalent to `duplicated` for integer vectors. Note the use of `seen.insert(x[i]).second` - `insert` returns a pair, the `first` value is an iterator that points to element and the `second` value is a boolean that's true if the value was a new addition to the set.

```
// [[Rcpp::export]]
LogicalVector duplicatedC(IntegerVector x) {
  std::tr1::unordered_set<int> seen;
  int n = x.size();
  LogicalVector out(n);

  for (int i = 0; i < n; ++i) {
    out[i] = !seen.insert(x[i]).second;
  }

  return out;
}
```

Map

A map is similar to a set, but instead of storing presence or absence, it can store additional data. It's useful for functions like `table` or `match` that need to look up a value. As with sets, there are ordered (`std::map`) and unordered (`std::tr1::unorded_map`) versions, but if output order matters it's usually faster to use an unordered map and sort the results.

Since maps have a value and a key, you need to specify both when initialising a map: `map<double, int>`, `unordered_map<int, double>`, and so on.

XXX: Insert example implementation of match when `String` complete

Exercises

To practice using the STL algorithms and data structures, implement the following using R functions in C++, using the hints provided:

- `median.default` using `partial_sort`
- `%in%` using `unordered_set` and the `find` or `count` methods
- `unique` using an `unordered_set` (challenge: do it in one line!)
- `min` using `std::min`, or `max` using `std::max`
- `which.min` using `min_element`, or `which.max` using `max_element`
- `setdiff`, `union` and `intersect` using sorted ranges and `set_union`, `set_intersection` and `set_difference`

Case studies

The following case studies illustrate some real life uses of C++ to replace slow R code.

Gibbs sampler

The following case study updates an example blogged about[5] by Dirk Eddelbuettel, illustrating the conversion of a gibbs sampler in R to C++. The R and C++ code shown below is very similar (it only took a few minutes to convert the R version to the C++ version), but runs about 20 times faster on my computer. Dirk's blog post also shows another way to make it even faster: using the faster (but presumably less accurate) random number generator functions in GSL (easily accessible from R through RcppGSL) can eke out another 2-3x speed improvement.

The R code is as follows:

```
gibbs_r <- function(N, thin) {
  mat <- matrix(nrow = N, ncol = 2)
  x <- y <- 0

  for (i in 1:N) {
```

[5] http://dirk.eddelbuettel.com/blog/2011/07/14/

```
    for (j in 1:thin) {
      x <- rgamma(1, 3, y * y + 4)
      y <- rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))
    }
    mat[i, ] <- c(x, y)
  }
  mat
}
```

This is straightforward to convert to C++. We:

- add type declarations to all variables
- use (instead of [to index into the matrix
- subscript the results of `rgamma` and `rnorm` to convert from a vector into a scalar

```
// [[Rcpp::export]]
NumericMatrix gibbs_cpp(int N, int thin) {
  NumericMatrix mat(N, 2);
  double x = 0, y = 0;

  for(int i = 0; i < N; i++) {
    for(int j = 0; j < thin; j++) {
      x = rgamma(1, 3, 1 / (y * y + 4))[0];
      y = rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))[0];
    }
    mat(i, 0) = x;
    mat(i, 1) = y;
  }

  return(mat);
}
```

Benchmarking the two implementations yields:

```
  library(microbenchmark)
  microbenchmark(
    gibbs_r(100, 10),
    gibbs_cpp(100, 10)
  )
```

R vectorisation vs. C++ vectorisation

This example is adapted from Rcpp is smoking fast for agent-based models in data frames[6]. The challenge is to predict a model response from three inputs. The basic R version looks like:

```
vacc1a <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * if (female) 1.25 else 0.75
  p <- max(0, p)
  p <- min(1, p)
  p
}
```

We want to be able to apply this function to many inputs, so we might write a vector-input version using a for loop.

```
vacc1 <- function(age, female, ily) {
  n <- length(age)
  out <- numeric(n)
  for (i in seq_len(n)) {
    out[i] <- vacc1a(age[i], female[i], ily[i])
  }
  out
}
```

If you're familiar with R, you'll have a gut feeling that this will be slow, and indeed it is. There are two ways we could attack this problem. If you have a good R vocabulary, you might immediately see how to vectorise the function (using `ifelse`, `pmin` and `pmax`). Alternatively, we could rewrite `vacc1a` and `vacc1` in C++, using our knowledge that loops and function calls have much lower overhead in C++.

Either approach is fairly straighforward. In R:

```
vacc2 <- function(age, female, ily) {
  p <- 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily
  p <- p * ifelse(female, 1.25, 0.75)
  p <- pmax(0, p)
  p <- pmin(1, p)
  p
}
```

[6]http://www.babelgraph.org/wp/?p=358

(If you've worked R a lot you might recognise some potential bottlenecks in this code: `ifelse`, `pmin`, and `pmax` are known to be slow, and could be replaced with `p + 0.75 + 0.5 * female`, `p[p < 0] <- 0`, `p[p > 1] <- 1`. You might want to try timing those variations yourself.)

Or in C++:

```
double vacc3a(double age, bool female, bool ily){
  double p = 0.25 + 0.3 * 1 / (1 - exp(0.04 * age)) + 0.1 * ily;
  p = p * (female ? 1.25 : 0.75);
  p = std::max(p, 0.0);
  p = std::min(p, 1.0);
  return p;
}

// [[Rcpp::export]]
NumericVector vacc3(NumericVector age, LogicalVector female, LogicalVector ily) {
  int n = age.size();
  NumericVector out(n);

  for(int i = 0; i < n; ++i) {
    out[i] = vacc3a(age[i], female[i], ily[i]);
  }

  return out;
}
```

We next generate some sample data, and check that all three versions return the same values:

```
n <- 1000
age <- rnorm(n, mean = 50, sd = 10)
female <- sample(c(T, F), n, rep = TRUE)
ily <- sample(c(T, F), n, prob = c(0.8, 0.2), rep = TRUE)

stopifnot(
  all.equal(vacc1(age, female, ily), vacc2(age, female, ily)),
  all.equal(vacc1(age, female, ily), vacc3(age, female, ily))
)
```

The original blog post forgot to do this, and hence introduced a bug in the C++ version: it used `0.004` instead of `0.04`. Finally, we can benchmark our three approaches:

```
microbenchmark(
  vacc1(age, female, ily),
  vacc2(age, female, ily),
  vacc3(age, female, ily))
```

Not surprisingly, our original approach with loops is very slow. Vectorising in R gives a huge speedup, and we can eke out even more performance (~10x) with the C++ loop. I was a little surprised that the C++ was so much faster, but it is because the R version has to create 11 vectors to store intermediate results, where the C++ code only needs to create 1.

Using Rcpp in a Package

The same C++ code that is used with `sourceCpp` can also be bundled into a package. There are several benefits of moving code from a standalone C++ source file to a package:

1. Your code can be made available to users without C++ development tools (at least on Windows or Mac OS X where binary packages are common)
2. Multiple source files and their dependencies are handled automatically by the R package build system
3. Packages provide additional infrastructure for testing, documentation and consistency

To generate a new Rcpp package that includes a simple hello, world function you can use the `Rcpp.package.skeleton` function as follows:

```
Rcpp.package.skeleton("NewPackage", attributes = TRUE)
```

To generate a package based on C++ files that you've been using with `sourceCpp` you can use the `cpp_files` parameter:

```
Rcpp.package.skeleton("NewPackage", example_code = FALSE,
                      cpp_files = c("convolve.cpp"))
```

Adding Rcpp to an existing package (Rcpp <= 0.10.6)

To add `Rcpp` to an existing package, you put your C++ files in the `src/` directory and modify/create the following configuration files:

- In `DESCRIPTION` add

```
LinkingTo: Rcpp
```

(If you're using advanced Rcpp features like modules, you'll also need 'Imports: Rcpp and to import the specific functions you're using)

- Make sure your `NAMESPACE` includes:

  ```
  useDynLib(mypackage)
  ```

- You need `src/Makevars` which contains:

  ```
  PKG_LIBS = `$(R_HOME)/bin/Rscript -e "Rcpp:::LdFlags()"`
  ```

  And `src/Makevars.win` that contains:

  ```
  PKG_LIBS = $(shell "${R_HOME}/bin${R_ARCH_BIN}/Rscript.exe" -e "Rcpp:::LdFlags()")
  ```

Adding Rcpp to an existing package (Rcpp >= 0.10.7)

To add `Rcpp` to an existing package, you put your C++ files in the `src/` directory and modify/create the following configuration files:

- In `DESCRIPTION` add

  ```
  LinkingTo: Rcpp
  Imports: Rcpp
  ```

- Make sure your `NAMESPACE` includes:

  ```
  useDynLib(mypackage)
  importFrom( Rcpp, sourceCpp )
  ```

We need to import something (anything) from Rcpp so that internal Rcpp code is properly loaded. In an ideal world, we would just need to use the `LinkingTo` but this is not enough.

More details

For more details see the Writing a package that uses Rcpp vignette[7].

If your packages uses the `Rcpp::export` attribute then one additional step in the package build process is required. The `compileAttributes` function scans the source files within a package for `Rcpp::export` attributes and generates the code required to export the functions to R.

You should re-run `compileAttributes` whenever functions are added, removed, or have their signatures changed. Note that if you build your package using RStudio or `devtools` then this step occurs automatically.

Learning more

More Rcpp

This chapter has only touched on a small part of Rcpp, giving you the basic tools to rewrite poorly performing R code in C++. Rcpp has many other capabilities that make it easy to interface R to existing C++ code, including:

- Additional features of attributes including specifying default arguments, linking in external C++ dependencies, and exporting C++ interfaces from packages. These features and more are covered in the Rcpp attributes[8] vignette.
- Automatically creating wrappers between C++ data structures and R data structures, including mapping C++ classes to reference classes. A good introduction to this topic is the vignette of Rcpp modules[9]
- The Rcpp quick reference guide[10] contains a useful summary of Rcpp classes and common programming idioms.

I strongly recommend keeping an eye on the Rcpp homepage[11] and signing up for the Rcpp mailing list[12]. Rcpp is still under active development, and is getting better with every release.

[7] http://cran.rstudio.com/web/packages/Rcpp/vignettes/Rcpp-package.pdf
[8] http://cran.rstudio.com/web/packages/Rcpp/vignettes/Rcpp-attributes.pdf
[9] http://cran.rstudio.com/web/packages/Rcpp/vignettes/Rcpp-modules.pdf
[10] http://cran.rstudio.com/web/packages/Rcpp/vignettes/Rcpp-quickref.pdf
[11] http://dirk.eddelbuettel.com/code/rcpp.html
[12] http://lists.r-forge.r-project.org/cgi-bin/mailman/listinfo/rcpp-devel

More C++

Writing performant code may also require you to rethink your basic approach: a solid understand of basic data structures and algorithms is very helpful here. That's beyond the scope of this book, but I'd suggest the "algorithm design manual"[13] or MIT's Introduction to Algorithms[14].

Other resources I've found helpful in learning C++ are:

- Effective C++[15] and Effective STL[16] by Scott Meyers.
- C++ Annotations[17], aimed at" knowledgeable users of C (or any other language using a C-like grammar, like Perl or Java) who would like to know more about, or make the transition to, C++"
- Algorithm Libraries[18], which provides a more technical, but still precise, description of important STL concepts. (Follow the links under notes)
- "Algorithms" by Robert Sedgewick and Kevin Wayne has a free online textbook[19] and a matching coursera course[20].

Acknowledgements

I'd like to thank the Rcpp-mailing list for many helpful conversations, particularly Romain Francois and Dirk Eddelbuettel who have not only provided detailed answers to many of my questions, but have been incredibly responsive at improving Rcpp. This chapter would not have been possible without JJ Allaire; he encouraged me to learn C++ and then answered many of my dumb questions along the way.

[13]http://amzn.com/0387948600?tag=devtools-20
[14]http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/
[15]http://amzn.com/0321334876?tag=devtools-20
[16]http://amzn.com/0201749629?tag=devtools-20
[17]http://www.icce.rug.nl/documents/cplusplus/cplusplus.html
[18]http://www.cs.helsinki.fi/u/tpkarkka/alglib/k06/
[19]http://algs4.cs.princeton.edu/home/
[20]https://www.coursera.org/course/algs4partI

Chapter 20

R's C interface

Introduction

Reading the source code of R is an extremely powerful technique for improving your R programming. However, at some point you will hit a brick wall: many R functions are implemented in C. This guide gives you a basic introduction to C and R's internal C api, giving you the basic knowledge needed to read the internals of R that are written in C.

If you want to write new high-performance code, we do not recommend using C, but instead strongly recommend using Rcpp to connect to C++. The Rcpp API protects you from many of the historical idiosyncracies of the R API, takes care of memory management for you, and provides many useful helper methods

The contents of this chapter are adapted from Section 5 ("System and foreign language interfaces") of Writing R extensions[1], focussing on best practices and modern tools. This means it does not cover:

- the `.C` interface
- the old api defined in `Rdefines.h`
- rarely used and esoteric language features

To understand existing C code, it's useful to generate simple examples of your own that you can experiment with. To that end, all examples in this chapter use the `inline` package, which makes it extremely easy to get up and running with C code. Make sure you have it installed and loaded with the following code:

[1] http://cran.r-project.org/doc/manuals/R-exts.html

```
install.packages("inline")
library(inline)
```

You'll also (obviously) need a working C compiler. Windows users can use Duncan Murdoch's Rtools[2]. Mac users will need the Xcode command line tools[3]. Most Linux distributions will come with the necessary compilers.

Differences between R and C

Even if you've never used C before, you should be able to read C code because the basic structure is similar to R. If you want to learn it more formally The C programming language[4] by Kernigan and Ritchie is a classic.

Important differences from R include:

- variables can only store specific types of object, and must be declared before use
- objects are modified in place, unless you specifically copy the object
- indices start at 0, not 1
- you must use a semi-colon at end of each expression
- you must have an explicit return statement
- assignment is done with `=`, not `<-`

Calling C functions from R

Generally, calling C functions from R involves two parts: a C function and an R function that uses `.Call`. The simple function below adds two numbers together and illustrates some of the important features of coding in C (creating new R vectors, coercing input arguments to the appropriate type and dealing with garbage collection).

```
// In C ----------------------------------------
#include <R.h>
#include <Rinternals.h>

SEXP add(SEXP a, SEXP b) {
  SEXP result;

  PROTECT(result = allocVector(REALSXP, 1));
```

[2]http://cran.r-project.org/bin/windows/Rtools/
[3]http://developer.apple.com/
[4]http://amzn.com/0131101633?tag=devtools-20

```
  REAL(result)[0] = asReal(a) + asReal(b);
  UNPROTECT(1);

  return(result);
}

# In R ----------------------------------------
add <- function(a, b) {
  .Call("add", a, b)
}
```

In this chapter we'll produce these two pieces in one step by using the `inline` package. This allows us to write:

```
add <- cfunction(signature(a = "integer", b = "integer"), "
  SEXP result;

  PROTECT(result = allocVector(REALSXP, 1));
  REAL(result)[0] = asReal(a) + asReal(b);
  UNPROTECT(1);

  return(result);
")
add(1, 5)
#> [1] 6
```

The C functions and macros that R provides for us to modify R data structures are all defined in the header file `Rinternals.h`. It's easiest to find and display this file from within R:

```
rinternals <- file.path(R.home(), "include", "Rinternals.h")
file.show(rinternals)
```

Before we begin writing and reading C code, we need to know a little about the basic data structures.

Basic data structures

At the C-level, all R objects are stored in a common datatype, the `SEXP`. (Technically, this is a pointer to a structure with typedef `SEXPREC`). A `SEXP` is a variant type, with subtypes for all R's data structures. The most important types are:

- `REALSXP`: numeric vectors

- `INTSXP`: integer vectors
- `LGLSXP`: logical vectors
- `STRSXP`: character vectors
- `VECSXP`: lists
- `CLOSXP`: functions (closures)
- `ENVSXP`: environments

Beware: At the C level, R's lists are `VECSXP`s not `LISTSXP`s. This is because early implementations of R used Lisp-like linked lists (now known as "pairlists") before moving to the S-like generic vectors that we now know as lists.

There are also `SEXP`s for less common object types:

- `CPLXSXP`: complex vectors
- `LISTSXP`: "pair" lists. At the R level, you only need to care about the distinction lists and pairlists for function arguments, but internally they are used in many more places.
- `DOTSXP`: '…'
- `SYMSXP`: names/symbols
- `NILSXP`: `NULL`

And `SEXP`s for internal objects, objects that are usually only created and used by C functions, not R functions:

- `LANGSXP`: language constructs
- `CHARSXP`: "scalar" strings (see below)
- `PROMSXP`: promises, lazily evaluated function arguments
- `EXPRSXP`: expressions

There's no built-in R function to easily access these names, but we can write one: (This is adapted from code in R's `inspect.c`)

```
sexp_type <- cfunction(c(x = "ANY"), '
  switch (TYPEOF(x)) {
    case NILSXP:      return mkString("NILSXP");
    case SYMSXP:      return mkString("SYMSXP");
    case LISTSXP:     return mkString("LISTSXP");
    case CLOSXP:      return mkString("CLOSXP");
    case ENVSXP:      return mkString("ENVSXP");
    case PROMSXP:     return mkString("PROMSXP");
    case LANGSXP:     return mkString("LANGSXP");
    case SPECIALSXP:  return mkString("SPECIALSXP");
    case BUILTINSXP:  return mkString("BUILTINSXP");
    case CHARSXP:     return mkString("CHARSXP");
    case LGLSXP:      return mkString("LGLSXP");
    case INTSXP:      return mkString("INTSXP");
    case REALSXP:     return mkString("REALSXP");
    case CPLXSXP:     return mkString("CPLXSXP");
    case STRSXP:      return mkString("STRSXP");
    case DOTSXP:      return mkString("DOTSXP");
    case ANYSXP:      return mkString("ANYSXP");
    case VECSXP:      return mkString("VECSXP");
    case EXPRSXP:     return mkString("EXPRSXP");
    case BCODESXP:    return mkString("BCODESXP");
    case EXTPTRSXP:   return mkString("EXTPTRSXP");
    case WEAKREFSXP:  return mkString("WEAKREFSXP");
    case S4SXP:       return mkString("S4SXP");
    case RAWSXP:      return mkString("RAWSXP");
    default:          return mkString("<unknown>");
}')
sexp_type(10)
#> [1] "REALSXP"
sexp_type(10L)
#> [1] "INTSXP"
sexp_type("a")
#> [1] "STRSXP"
sexp_type(T)
#> [1] "LGLSXP"
sexp_type(list(a = 1))
#> [1] "VECSXP"
sexp_type(pairlist(a = 1))
#> [1] "LISTSXP"
```

Character vectors

R character vectors are stored as `STRSXP`s, a vector type where every element is a `CHARSXP`. `CHARSXP`s are read-only objects and must never be modified. In particular, the C-style string contained in a `CHARSXP` should be treated as read-only; it's hard to do otherwise because the `CHAR` accessor function returns a `const char*`.

Strings have this more complicated design because individual `CHARSXP`'s (elements of a character vector) can be shared between multiple strings. This is an optimisation to reduce memory usage, and can result in unexpected behaviour:

```
x <- "banana"
y <- rep(x, 1e6)
object.size(x)
# 32-bit: 64 bytes
# 64-bit: 96 bytes
object.size(y) / 1e6
# 32-bit: 4.000056 bytes
# 64-bit: 8.000088 bytes
```

In 32-bit R, factors occupy about the same amount of memory as strings: both pointers and integers are 4 bytes. In 64-bit R, pointers are 8 bytes, so factors take about twice as much memory as strings.

Coercion and object creation

At the heart of every C function will be a set of conversions between R data structures and C data structures. Inputs and output will always be R data structures (`SEXP`s) and you will need to convert them to C data structures in order to do any work. An additional complication is the garbage collector: if you don't claim every R object you create, the garbage collector will think they are unused and delete them.

Object creation and garbage collection

The simplest way to create an new R-level object is `allocVector`, which takes two arguments, the type of `SEXP` (or `SEXPTYPE`) to create, and the length of the vector. The following code creates a three element list containing a logical vector, a numeric vector and an integer vector:

```
dummy <- cfunction(body = '
  SEXP vec, real, lgl, ints;

  PROTECT(real = allocVector(REALSXP, 2));
  PROTECT(lgl = allocVector(LGLSXP, 10));
  PROTECT(ints = allocVector(INTSXP, 10));

  PROTECT(vec = allocVector(VECSXP, 3));
  SET_VECTOR_ELT(vec, 0, real);
  SET_VECTOR_ELT(vec, 1, lgl);
  SET_VECTOR_ELT(vec, 2, ints);

  UNPROTECT(4);
  return(vec);
')
dummy()
#> [[1]]
#> [1] 8.179e-317 1.161e-316
```

```
#>
#> [[2]]
#>  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
#>
#> [[3]]
#>  [1] 15773840        0  6847160        0 11241728        0 14679416
#>  [8]        0 12431184        0
```

You might wonder what all the `PROTECT` calls do. They tell R that we're currently using each object, and not to delete it if the garbage collector is activated. (We don't need to protect objects that R already knows about, like function arguments).

You also need to make sure that every protected object is unprotected. `UNPROTECT` takes a single integer argument, n, and unprotects the last n objects that were protected. If your calls don't match, R will warn about a "stack imbalance in .Call".

Other specialised forms of `PROTECT` and `UNPROTECT` are needed in some circumstances: `UNPROTECT_PTR(s)` unprotects the object pointed to by the `SEXP` s, `PROTECT_WITH_INDEX` saves an index of the protection location that can be used to replace the protected value using `REPROTECT`. Consult the R externals section on garbage collection[5] for more details.

If you run `dummy()` a few times, you'll notice the output is basically random. This is because `allocVector` assigns memory to each output, but it doesn't clean it out first. For real functions, you'll want to loop through each element in the vector and zero it out. The most efficient way to do that is to use `memset`:

```
zeroes <- cfunction(c(n_ = "integer"), '
  int n = asInteger(n_);
  SEXP out;

  PROTECT(out = allocVector(INTSXP, n));
  memset(INTEGER(out), 0, n * sizeof(int));
  UNPROTECT(1);

  return out;
')
zeroes(10);
#>  [1] 0 0 0 0 0 0 0 0 0 0
```

Allocation shortcuts

There are a few shortcuts for allocating matrices and 3d arrays:

[5]http://cran.r-project.org/doc/manuals/R-exts.html#Garbage-Collection

```
allocMatrix(SEXPTYPE mode, int nrow, int ncol)
alloc3DArray(SEXPTYPE mode, int nrow, int ncol, int nface)
```

Beware `allocList` - it creates a pairlist, not a regular list.

The `mkNamed` function simplifies the creation of named vectors. The following code is equivalent to `list(a = NULL, b = NULL, c = NULL)`:

```
const char *names[] = {"a", "b", "c", ""};
mkNamed(VECSXP, names);
```

Extracting C vectors

There is a helper function for each atomic vector (apart from character, see following) that allows you to index into a `SEXP` and access the C-level data structure that lives at its heart.

The following example shows how to use the helper function `REAL` to inspect and modify a numeric vector:

```
add_one <- cfunction(c(x = "numeric"), "
  SEXP out;
  int n = length(x);

  PROTECT(out = allocVector(REALSXP, n));
  for (int i = 0; i < n; i++) {
    REAL(out)[i] = REAL(x)[i] + 1;
  }
  UNPROTECT(1);

  return out;
")
add_one(as.numeric(1:10))
#>  [1]  2  3  4  5  6  7  8  9 10 11
```

There are similar helpers for logical, `LOGICAL(x)`, integer, `INTEGER(x)`, complex `COMPLEX(x)` and raw vectors `RAW(x)`.

If you're working with long vectors, there's a performance advantage to using the helper function once and saving the result in a pointer:

```
add_two <- cfunction(c(x = "numeric"), "
  SEXP out;
  int n = length(x);
  double *px, *pout;
  PROTECT(out = allocVector(REALSXP, n));

  px = REAL(x);
  pout = REAL(out);
  for (int i = 0; i < n; i++) {
    pout[i] = px[i] + 2;
  }
  UNPROTECT(1);

  return out;
")
add_two(as.numeric(1:10))
#>  [1]  3  4  5  6  7  8  9 10 11 12

library(microbenchmark)
x <- as.numeric(1:1e6)
microbenchmark(
  add_one(x),
  add_two(x)
)
#> Unit: milliseconds
#>        expr   min    lq median    uq   max neval
#>  add_one(x) 5.652 6.276  6.660 7.712 13.93   100
#>  add_two(x) 4.586 5.253  5.595 6.051 11.75   100
```

On my computer, `add_two` is about twice as fast as `add_one` for a million element vector. This is a common idiom in the R source code.

Strings and lists are more complicated because the individual elements are `SEXP`s not C-level data structures. You can use `STRING_ELT(x, i)` and `VECTOR_ELT(x, i)` to extract individual components of strings and lists respectively. To get a single C string from a element in a R character vector, use `CHAR(STRING_ELT(x, i))`. Set values in a list or character vector with `SET_VECTOR_ELT` and `SET_STRING_ELT`.

Modifying strings

String vectors are a little more complicated. As discussed earlier, a string vector is a vector made up of pointers to immutable `CHARSXP`s, and it's the `CHARSXP` that contains the C string (which can be extracted using `CHAR`). The following function shows how to create a vector of fixed values:

```
abc <- cfunction(NULL, '
  SEXP out;
  PROTECT(out = allocVector(STRSXP, 3));

  SET_STRING_ELT(out, 0, mkChar("a"));
  SET_STRING_ELT(out, 1, mkChar("b"));
  SET_STRING_ELT(out, 2, mkChar("c"));

  UNPROTECT(1);

  return out;
')
abc()
#> [1] "a" "b" "c"
```

Things are a little harder if you want to modify the strings in the vector because you need to know a lot about string manipulation in C (which is hard, and harder to do right). For any problem that involves any kind of string modification, you're better off using Rcpp.

The following function just makes a copy of a string, so you can at least see how all the pieces work together.

```
copy <- cfunction(c(x = "character"), '
  SEXP out;
  int n = length(x);
  const char* letter;

  PROTECT(out = allocVector(STRSXP, n));
  for (int i = 0; i < n; i++) {
    letter = CHAR(STRING_ELT(x, i));
    SET_STRING_ELT(out, i, mkChar(letter));
  }
  UNPROTECT(1);

  return out;
')
copy(letters)
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
#> [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

One last useful function for operating with strings: we can use `STRING_PTR` to get a pointer to the `STRING_ELT`s in a vector, such that we can access the `STRING_ELT`s by de-referencing the pointer. Occasionally, this can be easier to work with. We'll show how this can make a simple example of reversing a vector of strings easier.

```
reverse <- cfunction( signature(x="character"), '
  SEXP out;
  int len = length(x);
  PROTECT( out = allocVector(STRSXP, len) );
  SEXP* out_ptr = STRING_PTR(out);
  SEXP* x_ptr = STRING_PTR(x);
  for( int i=0; i < len; ++i ) {
    out_ptr[i] = x_ptr[len-i-1];
  }
  UNPROTECT(1);
  return out;
')

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

Coercing scalars

There also a few helper functions if you want to turn the first element of an R vector into a C scalar:

- `asLogical(x): INTSXP -> int`
- `asInteger(x): INTSXP -> int`
- `asReal(x): REALSXP -> double`
- `CHAR(asChar(x)): STRSXP -> const char*`

And similarly it's easy to turn a C scalar into a length-one R vector:

- `ScalarLogical(x): int -> LGLSXP`
- `ScalarInteger(x): int -> INTSXP`
- `ScalarReal(x): double -> REALSXP`
- `mkString(x): const char* -> STRSXP`

These all create R-level objects, so need to be `PROTECT`ed.

Modifying objects

You must be very careful when modifying an object that the user has passed into the function. The following function has some very unexpected behaviour:

```
add_three <- cfunction(c(x = "numeric"), '
  REAL(x)[0] = REAL(x)[0] + 3;
  return(x);
')
x <- 1
y <- x
add_three(x)
#> [1] 4
x
#> [1] 4
y
#> [1] 4
```

Not only has it modified the value of `x`, but it has also modified `y`! This happens because of the way that R implements it's copy-on-modify philosophy. It does so lazily, so a complete copy only has to be made if you make a change: `x` and `y` point to the same object, and the object is only duplicated if you change either `x` or `y`.

To avoid problems like this, always `duplicate()` inputs before modifying them:

```
add_four <- cfunction(c(x = "numeric"), '
  SEXP x_copy;
  PROTECT(x_copy = duplicate(x));
  REAL(x_copy)[0] = REAL(x_copy)[0] + 4;
  UNPROTECT(1);
  return(x_copy);
')
x <- 1
y <- x
add_four(x)
#> [1] 5
x
#> [1] 1
y
#> [1] 1
```

Pairlists and symbols

R hides a few details of the underlying datastructures it uses. In some places in R code, it looks like you're working with a list (`VECSXP`), but behind the scenes R you're actually modifying a pairlist (`LISTSXP`). These include attributes, calls and ....

Pairlists differ from lists in the following ways:

- Pairlists are linked lists[6], a data structure which does not have any easy way to get to an arbitrary element of the list
- Pairlists have `tags`, not `names`, and tags are symbols, not strings.

Because you can't easily index into a specified location in a pairlist, R provides a set of helper functions to moved along the linked list. The basic helpers are `CAR` which extracts the first element of the list, and `CDR` which extracts the rest. These can be composed to get `CAAR`, `CDAR`, `CADDR`, `CADDR`, `CADDDR`. As well as the getters, R also provides `SETCAR`, `SETCDR` etc.

```
car <- cfunction(c(x = "ANY"), 'return(CAR(x));')
cdr <- cfunction(c(x = "ANY"), 'return(CDR(x));')
cadr <- cfunction(c(x = "ANY"), 'return(CADR(x));')

x <- quote(f(a = 1, b = 2))
car(x)
#> f
cdr(x)
#> $a
#> [1] 1
#>
#> $b
#> [1] 2
car(cdr(x))
#> [1] 1
cadr(x)
#> [1] 1
```

You can make new pairlists with `CONS` or `LCONS` (if you want a call object). A pairlist is always terminated with `R_NilValue`.

```
new_call <- cfunction(NULL, '
  SEXP out;

  out = LCONS(install("+"), LCONS(
      ScalarReal(10), LCONS(
        ScalarReal(5), R_NilValue)));
  return out;
')
new_call();
#> 10 + 5
```

Similarly, you can loop through all elements of a pairlist as follows:

[6] http://en.wikipedia.org/wiki/Linked_list

```
count <- cfunction(c(x = "ANY"), '
  SEXP el, nxt;
  int i = 0;

  for(nxt = x; nxt != R_NilValue; el = CAR(nxt), nxt = CDR(nxt)) {
    i++;
  }
  return(ScalarInteger(i));
')
count(quote(f(a, b, c)))
#> [1] 4
count(quote(f()))
#> [1] 1
```

`TAG` and `SET_TAG` allow you to get and set the tag (aka name) associated with an element of a pairlist. The tag should be a symbol. To create a symbol (the equivalent of `as.symbol` or `as.name` in R), use `install`.

Attributes are also pairlists behind the scenes, but come with the helper functions `setAttrib` and `getAttrib` to make access a little easier:

```
set_attr <- cfunction(c(obj = "ANY", attr = "character", value = "ANY"), '
  const char* attr_s = CHAR(asChar(attr));

  duplicate(obj);
  setAttrib(obj, install(attr_s), value);
  return(obj);
')
x <- 1:10
set_attr(x, "a", 1)
#>  [1]  1  2  3  4  5  6  7  8  9 10
#> attr(,"a")
#> [1] 1
```

There are some (confusingly named) shortcuts for common setting operations: `classgets`, `namesgets`, `dimgets` and `dimnamesgets` are the internal versions of the default methods of `class<-`, `names<-`, `dim<-` and `dimnames<-`.

```
tags <- cfunction(c(x = "ANY"), '
  SEXP el, nxt, out;
  int i = 0;

  for(nxt = CDR(x); nxt != R_NilValue; nxt = CDR(nxt)) {
    i++;
  }
  PROTECT(out = allocVector(VECSXP, i));

  for(nxt = CDR(x), i = 0; nxt != R_NilValue; i++, nxt = CDR(nxt)) {
    SET_VECTOR_ELT(out, i, TAG(nxt));
  }

  UNPROTECT(1);

  return out;
')
tags(quote(f(a = 1, b = 2, c = 3)))
#> [[1]]
#> a
#>
#> [[2]]
#> b
#>
#> [[3]]
#> c
tags(quote(f()))
#> list()
```

Missing and non-finite values

For floating point numbers, R's `NA` is a subtype of `NaN` so IEEE 754 arithmetic should handle it correctly. However, it is unwise to depend on such details, and is better to deal with missings explicitly:

- In `REALSXP`s, use the `ISNA` macro, `ISNAN`, or `R_FINITE` macros to check for missing, NaN or non-finite values. Use the constants `NA_REAL`, `R_NaN`, `R_PosInf` and `R_NegInf` to set those values
- In `INTSXP`s, compare/set values to `NA_INTEGER`
- In `LGLSXP`s, compare/set values to `NA_LOGICAL`
- In `STRSXP`s, compare/set `CHAR(STRING_ELT(x, i))` values to `NA_STRING`.

For example, a primitive implementation of `is.NA` might look like

```
is_na <- cfunction(c(x = "ANY"), '
  SEXP out;
  int n = length(x);
  PROTECT(out = allocVector(LGLSXP, n));

  for (int i = 0; i < n; i++) {
    switch(TYPEOF(x)) {
      case LGLSXP:
        LOGICAL(out)[i] = (LOGICAL(x)[i] == NA_LOGICAL);
        break;
      case INTSXP:
        LOGICAL(out)[i] = (INTEGER(x)[i] == NA_INTEGER);
        break;
      case REALSXP:
        LOGICAL(out)[i] = ISNA(REAL(x)[i]);
        break;
      case STRSXP:
        LOGICAL(out)[i] = (STRING_ELT(x, i) == NA_STRING);
        break;
      default:
        LOGICAL(out)[i] = NA_LOGICAL;
    }
  }
  UNPROTECT(1);

  return out;
')
is_na(c(NA, 1L))
#> [1]  TRUE FALSE
is_na(c(NA, 1))
#> [1]  TRUE FALSE
is_na(c(NA, "a"))
#> [1]  TRUE FALSE
is_na(c(NA, TRUE))
#> [1]  TRUE FALSE
```

It's worth noting that R's `base::is.na` returns `TRUE` for both `NA` and `NaN`s in a numeric vector, as opposed to the C level `ISNA` macro, which returns `TRUE` only for `NA_REAL`s.

There are a few other special values:

```
nil <- cfunction(NULL, 'return(R_NilValue);')
unbound <- cfunction(NULL, 'return(R_UnboundValue);')
missing_arg <- cfunction(NULL, 'return(R_MissingArg);')

x <- missing_arg()
```

```
x
#> Error:   "x",
```

Checking types in C

If the user provides different input to your function to what you're expecting (e.g. provides a list instead of a numeric vector), it's very easy to crash R. For this reason, it's a good idea to write a wrapper function that checks arguments are of the correct type, or coerces them if necessary. It's usually easier to do this at the R level. For example, going back to our first example of C code, we might rename it to `add_` and then write a wrapper function to check the inputs are ok:

```
add_ <- cfunction(signature(a = "integer", b = "integer"), "
  SEXP result;

  PROTECT(result = allocVector(REALSXP, 1));
  REAL(result)[0] = asReal(a) + asReal(b);
  UNPROTECT(1);

  return(result);
")
add <- function(a, b) {
  stopifnot(is.numeric(a), is.numeric(b), length(a) == 1, length(b) == 1)
  add_(a, b)
}
```

Or if we wanted to be more accepting of diverse inputs:

```
add <- function(a, b) {
  a <- as.numeric(a)
  b <- as.numeric(b)

  if (length(a) > 1) warning("Only first element of a used")
  if (length(a) > 1) warning("Only first element of b used")

  add_(a, b)
}
```

To coerce objects at the C level, use `PROTECT(new = coerceVector(old, SEXPTYPE))`. This will return an error if the `SEXP` can not be converted to the desired type. Note that these coercion functions do not use S3 dispatch.

To check if an object is of a specified type, you can use `TYPEOF`, which returns a `SEXPTYPE`:

```
is_numeric <- cfunction(c("x" = "ANY"), "
  return(ScalarLogical(TYPEOF(x) == REALSXP));
")
is_numeric(7)
#> [1] TRUE
is_numeric("a")
#> [1] FALSE
```

Or you can use one of the many helper functions. They all return 0 for FALSE and 1 for TRUE:

- For atomic vectors: `isInteger`, `isReal`, `isComplex`, `isLogical`, `isString`.
- For combinations of atomic vectors: `isNumeric` (integer, logical, real), `isNumber` (integer, logical, real, complex), `isVectorAtomic` (logical, interger, numeric, complex, string, raw)
- Matrices (`isMatrix`) and arrays (`isArray`)
- For other more esoteric object: `isEnvironment`, `isExpression`, `isList` (a pair list), `isNewList` (a list), `isSymbol`, `isNull`, `isObject` (S4 objects), `isVector` (atomic vectors, lists, expressions)

Note that some of these functions behave differently to the R-level functions with similar names. For example `isVector` is true for any atomic vector type, lists and expression, where `is.vector` is returns `TRUE` only if its input has no attributes apart from names.

Finding the C source code for a function

In many R functions you'll find code like `.Internal(mean(x))` or `.Primitive("sum")`. That means that most of the function is implemented at the C-level. There are two steps to finding the corresponding C source code:

- First, open src/main/names.c[7] and search for the name of the function. You'll find an entry that tells you the name of the function (which always starts with `do_`)
- Next, search the R source code for the name of that function. To make it easier to find where it's defined (rather than everywhere it's used), you can add `(SEXP`. e.g. to find the source code for `findInterval`, search for `do_findinterval(SEXP`.

[7] https://github.com/wch/r-source/blob/trunk/src/main/names.c

`.External`

An alternative to using `.Call` is to use `.External`. It is used almost identically, except that the C function will recieve a single arugment containing a `LISTSXP`, a pairlist from which the arguments can be extracted. This makes it possible to write functions that take a variable number of arguments.

`inline` does not currently support `.External` functions.

Using C code in a package

If you're putting your code in a package, it's generally a good idea to stop using `inline` and revert back to separate R and C functions. At a minimum, you'll need:

- R files in a `R/` directory
- C files in `src/` directory
- A `DESCRIPTION` in the main directory
- A `NAMESPACE` file containing `useDynLib(packagename)`, which can be generated using a `roxygen2` tag: `@useDynLib packagename`

Running `load_all(path/to/package)` will automatically compile and reload the code in your package.

Your C code will need these headers:

```
#include <R.h>
#include <Rinternals.h>
```

Part V

Packages

Chapter 21

Package development philosophy

This book espouses a particular philosophy of package development - it is not shared by all R developers, but it is one connected to a specific set of tools that makes package development as easy as possible.

There are three packages we will use extensively:

- `devtools`, which provides a set of R functions that makes package development as easy as possible.
- `roxygen2`, which translates source code comments into R's official documentation format
- `testthat`, which provides a friendly unit testing framework for R.

Other styles of package development don't use these packages, but in my experience they provide a useful trade off between speed and rigour. That's a theme that we'll see a lot in this chapter: base R provides rigorous tools that guarantee correctness, but tend to be slow. Sometimes you want to be able to iterate more rapidly and the tools we discuss will allow you to do so.

A package doesn't need to be complicated. You can start with a minimal subset of useful features and slowly build up over time. While there are strict requirements if you want to publish a package to the world on CRAN (and many of those requirements are useful even for your own packages), most packages won't end up on CRAN. Packages are really easy to create and use once you have the right set of tools.

Anytime you create some reusable set of functions you should put it in a package. It's the easiest path because packages come with conventions: you don't need to figure them out for yourself. You'll start with just your R code in the `R/` directory, and over time you can flesh it out with documentation (in `man/`), compiled code (in `src/`), data sets (in `data/`), and tests (in `inst/tests`).

Getting started

To get started, make sure you have the latest version of R: if you want to submit your work to CRAN, you'll need to make sure you're running all checks with the latest R.

You can install the packages you need for this chapter with:

```
install.packages("devtools", dependencies = TRUE)
```

You'll also need to make sure you have the appropriate development tools installed:

- On Windows, download and install Rtools: http://cran.r-project.org/bin/windows/Rtools/. This is not an R package.
- On Mac, make sure you have either XCode (free, available in the app store) or the "Command Line Tools for Xcode" (needs a free apple id, available from http://developer.apple.com/downloads)
- On Linux, make sure you've installed not only R, but the R development devtools. This a Linux package called something like `r-base-dev`.

You can check you have everything installed and working by running this code:

```
library(devtools)
has_devel()
```

It will print out some compilation code (this is needed to help diagnose problems), but you're only interested in whether it returns `TRUE` (everything's ok) or an error (which you need to investigate further).

Introduction to devtools

The goal of the devtools package is to make package development as painless as possible by encoding package building best practices in functions (so you don't have to remember or even know about them), and by minimising the iteration time when you're developing a package.

Most of the devtools functions we will use take a path to the package as their first argument. If the path is omitted, devtools will look in the current working directory - so for that reason, it's good practice to have your working directory set to the package directory.

The functions that you'll use most often are those that facilitate the [[package development cycle |development]]:

- `load_all()`: simulates package installation and loading by `source()`ing all files in the `R/` directory, compiling and linking C, C++ and Fortran files in the `src/` and `load()`ing data files in the `data/` directory. More on that in [[package development |development]]
- `document()`: extracts documentation from source code comments and creates `Rd` files in the `man/` directory. You can use `dev_help()` and `dev_example()` instead of `help()` and `example()` to preview these files without installing the package. More on that in [[documentation]].
- `test()`: runs all unit tests in the `inst/tests/` directory and reports the results. More on that in [[testing]].

Other functions mimic standard R commands that you run from the command line:

- `build()` is equivalent to `R CMD build` and bundles package. See [[package-basics]] for more about what a bundled package is.
- `install()` is equivalent to `R CMD INSTALL` and installs a package into a local library. Learn more about the installation process in [[package-basics]].
- `check()` is equivalent to `R CMD check`, and runs a large set of automated tests against your package. Read more about checking as part of the [[release]] process.

These tools should be more reliable than running the equivalent commands in the terminal (and much easier to use if you're not familiar with the terminal). They do more to ensure that command-line R is running in exactly the same way as your R GUI. They check that you're running the same version of R, with the same library paths, and with a standard collation order. These are things you don't need to worry about most of the time, but if they ever trip you up, it can take hours to figure out the source of the problem. `check()` and `install()` also run build first, which is recommended best practice.

There are two other functions that you use less commonly, only at the start and the end of package development:

- `create()` creates a new package, and fills it out with the basics (this is the `devtools` equivalent of `package.skeleton`)
- `release()` checks the package, checks that you've done what the CRAN maintainers expect, uploads to CRAN and drafts an email for the CRAN maintainers

Chapter 22

Package basics

An R package is the basic unit of reusable code. You need to master the art of making R packages if you want others to use your code. At their heart, packages are quite simple and only have two essential components:

- a `DESCRIPTION` file that describes the package, what it does, who's allowed to use it (the license) and who to contact if you need help
- an `R` directory that contains your R code.

If you want to distribute R code to someone else, there's no excuse not to use a simple package: it's a standard structure, and you can easily build it out into a complete package by adding documentation, data and tests.

This document explains how to get started, with a description of package structure, tips for naming your package, and more details about the `DESCRIPTION` file.

The most accurate resource for up-to-date details on package development is always the official writing R extensions[1] guide. However, it's rather hard to read and follow if you're not already familiar with the basics of packages. It's also exhaustive, covering every possible package component, rather than focussing on the most common and useful components as this package does. Once you are familiar with the content here, you should find R extensions a little easier to read.

Package essentials

As mentioned above, there are only two elements that you must have:

[1] http://cran.r-project.org/doc/manuals/R-exts.html#Creating-R-packages

- the `DESCRIPTION` file, which provides metadata about the package, and is described in the following section.
- the `R/` directory where your R code lives (in `.R` or `.r` files).

If you don't want to create this by hand, you can use `devtools::create` which initialises the directory structure and includes a few other files that most packages have.

Optional components

Almost all R packages also have:

- the `man/` directory where your [[function documentation|documenting-functions]]. In the style of package development described in this book, you'll never personally touch the files in this directory. Instead, they will be automatically generated from comments in your source code using the `roxygen2` package

After the code and function documentation, the most important optional components of an R package help your users learn how to use your package. The following files and directories are described in more detail in [[documenting packages]].

- the `NEWS` file describes the changes in each version of the package. Using the standard R format will allow you to take advantage of many automated tools for displaying changes between versions.
- the `README` file gives a general overview of your package, including why it's important. This text should be included in any package announcement, to help others understand why they might want to use your package.
- the `inst/CITATION` file describes how to cite your package. If you have published a peer reviewed article which you'd like people to cite when they use your software, this is the place to put it.
- the `demo/` directory contains larger scale demos, that use many features of the package.
- the `inst/doc/` directory is used for larger scale documentation, like vignettes, long-form documents which show how to combine multiple parts of your package to solve problems.

Other optional files and directories are part of good development practice:

- a `NAMESPACE` file describes which functions are part of the formal API of the package and are available for others to use. See [[namespaces]] for more details.
- `tests/` and `inst/tests/` contains [[unit tests|testing]] which ensure that your package is operating as designed. In this book, we'll focus on the `testthat` package for writing tests.
- the `data/` directory contains `.rdata` files, used to include sample datasets (or other R objects) with your package.

There are other directories that we won't cover. You might see these in other packages you download from CRAN, but these topics are outside the scope of this book.

- `src/`: C, C++ and fortran source code
- `exec/`: executable scripts
- `po/`: translation files

Getting started

When creating a package the first thing (and sometimes the most difficult) is to come up with a name for it. There's only one formal requirement:

- The package name can only consist of letters and numbers, and must start with a letter.

But I have a few additional recommendations:

- Make the package name googleable, so that if you google the name you can easily find it. This makes it easy for potential users to find your package, and it's also useful for you, because it makes it easier to find out who is using it.
- Avoid using both upper and lower case letters: they make the package name hard to type and hard to remember. For example, I can never remember if it's `Rgtk2` or `RGTK2` or `RGtk2`.

Some strategies I've used in the past to create packages names:

- Use abbreviations: `lvplot` (letter value plots), `meifly` (models explored interactively)

- Add an extra R: `stringr` (string processing), `tourr` (grand tours), `httr` (HTTP requests), `helpr` (alternative documentation view)
- Find a name evocative of the problem and modify it so that it's unique: `plyr` (generalisation of apply tools), `lubridate` (makes dates and times easier), `mutatr` (mutable objects), `classifly` (high-dimensional views of classification)

Once you have a name, create a directory with that name, and inside that create an `R` subdirectory and a `DESCRIPTION` file (note that's there's no extension, and the file name must be all upper case).

The `R/` directory

The `R/` directory contains all your R code, so copy in any existing code.

It's up to you how you arrange your functions into files. There are two possible extremes: all functions in one file, and each function in its own file. I think these are both too extreme, and I suggest grouping related functions into a single file. My rule of thumb is that if I can't remember which file a function lives in, I probably need to split them up into more files: having only one function in a file is perfectly reasonable, particularly if the functions are large or have a lot of documentation. As you'll see in the next chapter, often the code for the function is small compared to its documentation (it's much easier to do something than it is to explain to someone else how to do it.)

The next step is to create a `DESCRIPTION` file that defines package metadata.

A minimal `DESCRIPTION` file

A minimal description file (this one is taken from an early version of plyr) looks like this:

```
Package: plyr
Title: Tools for splitting, applying and combining data
Description:
Version: 0.1
Author: Hadley Wickham <h.wickham@gmail.com>
Maintainer: Hadley Wickham <h.wickham@gmail.com>
License: MIT
```

This is the critical subset of package metadata: what it's called (`Package`), what it does (`Title`, `Description`), who's allowed to use and distribute it (`License`), who wrote it (`Author`), and who to contact if you have problems (`Maintainer`). Here I've left the `Description` blank to illustrate that if you haven't decided what the correct value is yet, it's ok to leave it blank.

Again, the six required elements are:

- `Package`: name of the package. Should be the same as the directory name.
- `Title`: a one line description of the package.
- `Description`: a more detailed paragraph-length description.
- `Version`: the version number, which should be of the the form `major.minor.patchlevel`. See `?package_version` for more details on the package version formats. I recommended following the principles of semantic versioning[2].
- `Maintainer`: a single name and email address for the person responsible for package maintenance.
- `License`: a standard abbreviation for an open source license, like `GPL-2` or `BSD`. A complete list of possibilities can be found by running `file.show(file.path(R.home(), "share/licenses/license.db"))`. If you are using a non-standard license, put `file LICENSE` and then include the full text of the license in a `LICENSE`.

Other `DESCRIPTION` components

A more complete `DESCRIPTION` (this one from a more recent version of `plyr`) looks like this:

```
Package: plyr
Title: Tools for splitting, applying and combining data
Description: plyr is a set of tools that solves a common set of
    problems: you need to break a big problem down into manageable
    pieces, operate on each pieces and then put all the pieces back
    together.  For example, you might want to fit a model to each
    spatial location or time point in your study, summarise data by
    panels or collapse high-dimensional arrays to simpler summary
    statistics. The development of plyr has been generously supported
    by BD (Becton Dickinson).
URL: http://had.co.nz/plyr
Version: 1.3
Maintainer: Hadley Wickham <h.wickham@gmail.com>
```

[2]http://semver.org/

```
Author: Hadley Wickham <h.wickham@gmail.com>
Depends: R (>= 2.11.0)
Suggests: abind, testthat (>= 0.2), tcltk, foreach
Imports: itertools, iterators
License: MIT
```

This `DESCRIPTION` includes other components that are optional, but still important:

- `Depends`, `Suggests`, `Imports` and `Enhances` describe which packages this package needs. They are described in more detail in [[namespaces]].
- `URL`: a url to the package website. Multiple urls can be separated with a comma or whitespace.

Instead of `Maintainer` and `Author`, you can `Authors@R`, which takes a vector of `person()` elements. Each person object specifies the name of the person and their role in creating the package:

- `aut`: full authors who have contributed much to the package
- `ctb`: people who have made smaller contributions, like patches.
- `cre`: the package creator/maintainer, the person you should bother if you have problems

Other roles are listed in the help for person. Using `Authors@R` is useful when your package gets bigger and you have multiple contributors that you want to acknowledge appropriately. The equivalent `Authors@R` syntax for plyr would be:

```
Authors@R: person("Hadley", "Wickham", role = c("aut", "cre"))
```

There are a number of other less commonly used fields like `BugReports`, `KeepSource`, `OS_type` and `Language`. A complete list of the `DESCRIPTION` fields that R understands can be found in the R extensions manual[3].

Source, binary and bundled packages

So far we've just described the structure of a source package: the development version of a package that lives on your computer. There are also two other types of package: bundled packages and binary packages.

[3] http://cran.r-project.org/doc/manuals/R-exts.html#The-DESCRIPTION-file

A package **bundle** is a compressed version of a package in a single file. By convention, package bundles in R use the extension `.tar.gz`. This is Linux convention indicating multiple files have been collapsed into a single file (`.tar`) and then compressed using gzip (`.gz`). The package bundle is useful if you want to manually distribute your package to another R package developer. It is not OS specific. You can use `devtools::build()` to make a package bundle.

If you want to distribute your package to another R user (i.e. someone who doesn't necessarily have the development tools installed) you need to make a **binary** package. Like a package bundle, a binary package is a single file, but if you uncompress it, you'll see that the internal structure is a little different to a source package:

- a `Meta/` directory contains a number of `Rds` files. These contain cached metadata about the package, like what topics the help files cover and parsed versions of the `DESCRIPTION` files. (If you want to look at what's in these files you can use `readRDS`)
- a `html/` directory contains some files needed for the HTML help.
- there are no `.R` files in the `R/` directory - instead there are three files that store the parsed functions in an efficient format. This is basically the result of loading all the R code and then saving the functions with `save`, but with a little extra metadata to make things as fast as possible.
- If you had any code in the `src/` directory there will now be a `libs/` directory that contains the results of compiling that code for 32 bit (`i386/`) and 64 bit (`x64`)

Binary packages are platform specific: you can't install a Windows binary package on a Mac or vice versa. You can use `devtools::build(binary = TRUE)` to make a package bundle.

An **installed** package is just a binary package that's been uncompressed into a package library, described next.

Package libraries

A library is a collection of installed packages. You can have multiple libraries on your computer and most people have at least two: one for the recommended packages that come with a base R install (like `base`, `stats` etc), and one library where the packages you've installed live. The default is to make that directory dependent on which version of R you have installed - that's why you normally lose all your packages when you reinstall R. If you want to avoid this behaviour, you can manually set the `R_LIBS` environmental variable to point somewhere else. `.libPaths()` tells you where your current libraries live.

When you use `library(pkg)` to load a package, R looks through each path in `.libPaths()` to see if a directory called `pkg` exists.

Installing packages

Package installation is the process whereby a source package gets converted into a binary package and then installed into your local package library. There are a number of tools that automate this process:

- `install.packages()` installs a package from CRAN. Here CRAN takes care of making the binary package and so installation from CRAN basically is equivalent to downloading the binary package value and unzipping it in `.libPaths()[1]` (but you should never do this by hand because the process also does other checks)
- `devtools::install()` installs a source package from a directory on your computer.
- `devtools::install_github()` installs a package that someone has published on their github[4] account. There are a number of similar functions that make it easy to install packages from other internet locations: `install_url`, `install_gitorious`, `install_bitbucket`, and so on.

Exercises

(to be integrated throughout the chapter)

- Go to CRAN and download the source and binary for XXX. Unzip and compare. How do they differ?
- Download the **source** packages for XXX, YYY, ZZZ. What directories do they contain?
- Where is your default library? What happens if you install a new package from CRAN?

[4] http://github

Chapter 23

The package development cycle

The package development cycle describes the sequence of operations that you use when developing a package. You probably already have a sequence of operations that you're already comfortable with when developing a single file of R code. It might be:

- Try something out on the command line.
- Modify it until it works and then copy and paste the command into an R file.
- Every now and then restart R and source in the R file to make sure you haven't missed anything.

Or:

- Write all your functions in an R file.
- `source()` the file into your current session.
- Interactively try out the functions and see if they return the correct results.
- Repeat the above steps until the functions work the way you expect.

Things get a bit harder when you're working on a package, because you have multiple R files. You might also be a little bit more worried about checking that your functions work, not only now, on your computer, but also in the future and on other computers.

The sections below describe useful practices for developing packages, where you might have many R files, and you are also thinking about documentation and testing. Both are supported by the `devtools` function `dev_mode`, which helps to keep your production and development packages separate.

Dev mode

When you're developing packages, typically you are developing them because you're using them - and that easily leads to confusion. When you're doing data analysis, you want to use a stable version but when you are developing code, you want to make sure that you can test changes that you have made.

The `dev_mode()` function makes it easier to manage the distinction between stable and development versions of a package. During development, the function `dev_mode()` will install the package in a special library that is not part of the default library path. Here, you can install and test new packages. During analysis, you keep `dev_mode()` off and R won't find the development packages:

```
# During development
dev_mode()
install("mypackage")
library(mypackage) # uses the development version

# Fresh R session
library(mypackage) # uses the released package you've installed previously
```

Key functions

The three functions that you'll use most often are:

- `load_all("pkg")`, which loads code, data, and C files. These are loaded into a non-global environment to avoid conflicts and so all functions can easily be removed. By default `load_all` will only load changed files to save time: if you want to reload everything from scratch, run `load_all("pkg", T)`
- `test("pkg")` runs all tests in `inst/tests/` and reports the results.
- `document("pkg")` runs `roxygen` on the package to update all documentation.

This makes the development process very easy. After you've modified the code, you run `load_all("pkg")` to load it in to R. You can explore it interactively from the command line, or type `test("pkg")` to run all your automated tests.

Some GUIs have integrated supported for these functions. Rstudio provides the build menu, which allows you to easily run `load_all` with a single key press (Ctrl + Shift + L at time of writing). ESS?

The following sections describe how you might combine these functions into a process.

Development cycles

It's useful to distinguish between exploratory programming and confirmatory programming (in the same sense as exploratory and confirmatory data analysis), because the development cycle differs in several important ways.

Confirmatory programming

Confirmatory programming happens when you know what you need to do and what the results of your changes will be (new feature X appears or known bug Y disappears); you just need to figure out the way to do it. Confirmatory programming is also known as test driven development[1] (TDD), a development style that grew out of extreme programming. The basic idea is that, before you implement any new feature or fix a known bug, you should:

1. Write an automated test and run `test()` to make sure the test fails (so you know you've captured the bug correctly).
2. Modify code to fix the bug or implement the new feature.
3. Run `test(pkg)` to reload the package and re-run the tests.
4. Repeat 2–4 until all tests pass.
5. Update documentation comments, run `document()`, and update `NEWS`.

For this paradigm, you might also want to use `testthat::auto_test()`, which will watch your tests and code and will automatically rerun your tests when either changes. This allows you to skip step three: you just modify your code and watch to see if the tests pass or fail.

Exploratory programming

Exploratory programming is the complement of confirmatory programming, when you have some idea of what you want to achieve, but you're not sure

[1] http://en.wikipedia.org/wiki/Test-driven_development

about the details. You're not sure what the functions should look like, what arguments they should have and what they should return. You may not even be sure how you are going to break down the problem into pieces. In exploratory programming, you're exploring the solution space by writing functions and you need the freedom to rewrite large chunks of the code as you understand the problem domain better.

The exploratory programming cycle is similar to confirmatory, but it's not usually worth writing the tests before writing the code, because the interface will change so much:

1. Edit code and reload with `load_all()`.
2. Test interactively.
3. Repeat 1–2 until code works.
4. Write automated tests and `test()`.
5. Update documentation comments, run `document()`, and update `NEWS`

The automated tests are still vitally important because they are what will prevent your code from failing silently in the future.

Chapter 24

Package level documentation

Package level documentation exists to help the user understand how to use the package as a whole. Function level documentation is useful once you know exactly what you want to do, and you've discovered the appropriate function: once you've done all that, function-level documentation helps you use the function appropriately. Package-level documentation gets you to point where you can do that: it explains what the package does as a whole, breaks the functions down into useful categories, and shows you how to combine the functions (and maybe functions from other packages) to solve problems.

There are XXX components to package level documentation. None of them are compulsory, but the more you make, the easier it will be for others to use your package.

- `README`, `NEWS`, `CITATION` are specially formatted plain-text documentation that provide a brief overview of your package, what's changed recently, and how to cite your package.
- Vignettes provide long form pdf documentation
- Demos are pure R files that provide case studies linking together multiple components of the package
- It's also a good idea to include "function" documentation for your package - that's described in the following chapter.

README

The `README` file lives in the package directory. It should be fairly short (3-4 paragraphs) and answer the following questions:

- Why should someone use your package?
- How does it compare to other existing solutions?
- What are the main functions?

If you're using github, this will appear on the package home page. I also recommend using it when you announce a new version of your package.

Some examples from our packages follow. Note that most of these use markdown, http://daringfireball.net/projects/markdown/, a plain text formatting language to add headings, basic text formatting and bullets. A brief introduction to markdown is included in the appendix. If you use markdown, you should call you readme file `README.md`.

plyr

```
plyr is a set of tools for a common set of problems: you need to __split__
up a big data structure into homogeneous pieces, __apply__ a function to
each piece and then __combine__ all the results back together. For
example, you might want to:

  * fit the same model each patient subsets of a data frame
  * quickly calculate summary statistics for each group
 * perform group-wise transformations like scaling or standardising

It's already possible to do this with base R functions (like split and the
apply family of functions), but plyr makes it all a bit easier with:

  * totally consistent names, arguments and outputs
  * convenient parallelisation through the foreach package
  * input from and output to data.frames, matrices and lists
  * progress bars to keep track of long running operations
  * built-in error recovery, and informative error messages
  * labels that are maintained across all transformations

Considerable effort has been put into making plyr fast and memory
efficient, and in many cases plyr is as fast as, or faster than, the
built-in equivalents.

A detailed introduction to plyr has been published in JSS: "The
```

```
Split-Apply-Combine Strategy for Data Analysis",
http://www.jstatsoft.org/v40/i01/. You can find out more at
http://had.co.nz/plyr/, or track development at
http://github.com/hadley/plyr. You can ask questions about plyr (and data
manipulation in general) on the plyr mailing list. Sign up at
http://groups.google.com/group/manipulatr.
```

```
stringr

Strings are not glamorous, high-profile components of R, but they do play
a big role in many data cleaning and preparations tasks. R provides a
solid set of string operations, but because they have grown organically
over time, they can be inconsistent and a little hard to learn.
Additionally, they lag behind the string operations in other programming
languages, so that some things that are easy to do in languages like Ruby
or Python are rather hard to do in R. The `stringr` package aims to remedy
these problems by providing a clean, modern interface to common string
operations.

More concretely, `stringr`:

 * Processes factors and characters in the same way.

 * Gives functions consistent names and arguments.

 * Simplifies string operations by eliminating options that you don't need
   95% of the time.

 * Produces outputs than can easily be used as inputs. This includes
   ensuring that missing inputs result in missing outputs, and zero length
   inputs result in zero length outputs.

 * Completes R's string handling functions with useful functions from
   other programming languages.
```

NEWS

The NEWS file should list all changes that have occurred since the last release of the package.

The following sample shows the NEWS file from the stringr package.

```
stringr 0.5
===========
* new `str_wrap` function which gives `strwrap` output in a more
  convenient format

* new `word` function extract words from a string given user defined
  separator (thanks to suggestion by David Cooper)

* `str_locate` now returns consistent type when matching empty string
  (thanks to Stavros Macrakis)

* new `str_count` counts number of matches in a string.

* `str_pad` and `str_trim` receive performance tweaks - for large vectors
  this should give at least a two order of magnitude speed up

* str_length returns NA for invalid multibyte strings

* fix small bug in internal `recyclable` function
```

`NEWS` has a special format, but it's not well documented. The basics are:

- The information for each version should start with the name of the package and its version number, followed by a line of `=`s.
- Each change should be listed with a bullet. If a bullet continues over multiple lines, the second and subsequent lines need to be indented by at least two spaces. (I usually add a blank line between each bullet to make it easier to read.)
- If you have many changes, you can use subheadings to divide them into sections. A subheading should be all upper case and flush left.
- I use markdown formatting inside the bullets. This doesn't help the formatting in R, but is useful if you want to publish the `NEWS` file elsewhere.

You can use `devtools::show_news()` to display the `NEWS` using R's built-in parser and check that it appears correctly. `show_news()` defaults to showing just the news for the most recent version of the package. You can override this by using argument `latest = FALSE`.

`CITATION`

The `CITATION` file lives in the `inst` directory and is intimately connected to the `citation()` function which tells you how to cite R and R packages. Calling `citation()` without any arguments tells you how to cite base R:

```
To cite R in publications use:

  R Core Team (2012). R: A language and environment for statistical
  computing. R Foundation for Statistical Computing, Vienna, Austria.
  ISBN 3-900051-07-0, URL http://www.R-project.org/.

A BibTeX entry for LaTeX users is

  @Manual{,
    title = {R: A Language and Environment for Statistical Computing},
    author = {{R Core Team}},
    organization = {R Foundation for Statistical Computing},
    address = {Vienna, Austria},
    year = {2012},
    note = { {ISBN} 3-900051-07-0},
    url = {http://www.R-project.org/},
  }

We have invested a lot of time and effort in creating R, please cite it
when using it for data analysis. See also 'citation("pkgname")' for
citing R packages.
```

This is generated from a `CITATION` file that looks like this:

```
bibentry("Manual",
   title = "R: A Language and Environment for Statistical Computing",
   author = person("R Core Team"),
   organization = "R Foundation for Statistical Computing",
   address      = "Vienna, Austria",
   year   = version$year,
   note   = "{ISBN} 3-900051-07-0",
   url    = "http://www.R-project.org/",

   mheader = "To cite R in publications use:",

   mfooter =
    paste("We have invested a lot of time and effort in creating R,",
      "please cite it when using it for data analysis.",
      "See also", sQuote("citation(\"pkgname\")"),
      "for citing R packages.", sep = " ")
)
```

As you can see, it's pretty simple: you only need to learn one new function, `bibentry()`. The most important arguments, are `bibtype` (the first argument, which can "Article", "Book", "PhDThesis" and so on), and then the standard bibliographic information like `title,`, `author`, `year`, `publisher`, `journal`, `volume`, `issue`, `pages` and so on (they are all described in detail in `?bibEntry`). The header (`mheader`) and footer (`mfooter`) are optional, and are useful places for additional exhortations.

Vignettes

Vignettes are long-form pdf guides to your package. They are sweave documents that live in the `vignettes/` directory. To write a vignette you must be familiar with both Sweave and LaTeX. Describing these is outside the scope of this book, but some useful resources are:

- http://biostat.mc.vanderbilt.edu/wiki/Main/SweaveLatex
- Resource 2
- Resource 3

If you write a nice vignette, you might want to consider submitting it to the Journal of Statistical Software or the R Journal. Both are electronic only journals and peer-reviewing can be very helpful for improving the quality of your vignette and the related software.

Demos

A demo is very much like a function example, but is longer, and shows how to use multiple functions together. Demos are `.R` files that live in the `demo/` package directory, and are accessed with the `demo()` function.

(NOT YET IMPLEMENTED) The `demos` directory also needs an index. The easiest way to generate that index is to add a roxygen comment with `@demoTitle` tag:

```
#' @demoTitle my title
```

The roxygen process that turns this comment into an index is described in the next chapter.

Chapter 25

Documenting functions

Documentation is one of the most important aspects of good code. Without it, users won't know how to use your package, and are unlikely to do so. Documentation is also useful for you in the future (so you remember what the heck you were thinking!), and for other developers working on your package.

The standard way to write R documentation is to create `.Rd` files in the `man/` directory. These files describe each object (function, data set, class, generic or method) in your package, documenting how it works and giving the user examples of how to use it. Rather than writing Rd files by hand, we're going to use the roxygen2[1] package. With roxygen, you write the documentation in comments next to each function, and then run an R script to create the main files.

This workflow has a number of advantages over writing `.Rd` files by hand:

- Code and documentation are adjacent so when you modify your code, it's easy to remember that you need to update the documentation.
- Roxygen2 dynamically inspects the objects that are documenting, so it can figure out a lot of information `.Rd` files need by itself. For example, it can automatically add links to super and subclasses.
- It takes care of other files that are fiddly or downright painful to maintain by hand: the namespace[2], collate order in description, and the demos index.
- Abstracts over the differences in documenting S3 and S4 methods, generics and classes so that they behave basically the same.

[1] http://roxygen.org/
[2] ‘NAMESPACE‘

Note: this chapter currently describes the use of the in-development roxygen3[3] package. This will be merged back into roxygen2 in the near future.

This chapter will proceed as follows. First, we'll discuss the basic documentation process, what each step does, and how to see the output at every stage. Next we'll show you, at a high-level, how to document all the different types of R objects (functions, datasets, s3 methods, s4 methods, s4 classes, r5 classes, and packages). After that you'll learn how to format text within the documentation.

Help

You've probably used help a lot, but you might not be aware of the more advanced features:

- `package?lubridate`
- `class?myclass`
- `methods?`
- `method?`
- `method?combo("numeric", "numeric")`
- `?combo(1:10, letters)`

Roxygen automatically takes care of generating the special aliases needed to make these lookups work.

How does help work? It finds the matching Rd file, and compiles it to either text or HTML output. It's complicated by the fact that binary packages don't include individual Rd files, they actually included a pre-parsed database of Rd files.

Roxygen process

There's a three step process to go from the R comments in the source files to Rd files to human readable documentation. The process starts with special comments starting with `#'` that indicate a comment is a roxygen comment:

```
#' Order a data frame by its columns.
#'
#' This function completes the subsetting, transforming and ordering triad
#' with a function that works in a similar way to \code{\link{subset}} and
#' \code{\link{transform}} but for reordering a data frame by its columns.
#' This saves a lot of typing!
```

[3] http://github.com/hadley/roxygen3

```
#'
#' @param df data frame to reorder
#' @param ... expressions evaluated in the context of \code{df} and
#'   then fed to \code{\link{order}}
#' @keywords manip
#' @export
#' @examples
#' mtcars[with(mtcars, order(cyl, disp)), ]
#' arrange(mtcars, cyl, disp)
#' arrange(mtcars, cyl, desc(disp))
arrange <- function(df, ...) {
  ord <- eval(substitute(order(...)), df, parent.frame())
  unrowname(df[ord, ])
}
```

To convert roxygen comments to the official `.Rd` files, we'll use `devtools::document()`. Like the other devtools functions you've seen so far, it takes a package directory as its first argument, and if you omit it, it will use the current working directory.

This produces an `.Rd` file in the `man/` directory:

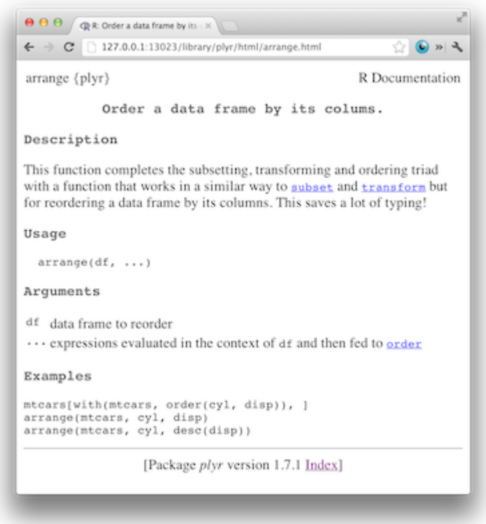
```
\name{arrange}
\alias{arrange}
\title{Order a data frame by its columns.}
\usage{arrange(df, ...)}
\description{
  This function completes the subsetting, transforming and
  ordering triad with a function that works in a similar
  way to \code{\link{subset}} and \code{\link{transform}}
  but for reordering a data frame by its columns. This
  saves a lot of typing!
}
\keyword{manip}
\arguments{
  \item{df}{data frame to reorder}
  \item{...}{expressions evaluated in the context of
    \code{df} and then fed to \code{\link{order}}}
}
\examples{mtcars[with(mtcars, order(cyl, disp)), ]
arrange(mtcars, cyl, disp)
arrange(mtcars, cyl, desc(disp))}
```

Rd files are a special file format loosely based on LaTeX. You can read more about the Rd format in the R extensions[4] manual. We'll avoid discussing Rd

[4] http://cran.r-project.org/doc/manuals/R-exts.html#Rd-format

files as much as possible, focussing instead on what you need to know about roxygen.

When you request documentation, R then converts this file into the display you're probably used to seeing:


R: Order a data frame by its
127.0.0.1:13023/library/plyr/html/arrange.html

arrange {plyr} R Documentation

Order a data frame by its colums.

Description

This function completes the subsetting, transforming and ordering triad with a function that works in a similar way to subset and transform but for reordering a data frame by its columns. This saves a lot of typing!

Usage

```
arrange(df, ...)
```

Arguments

df	data frame to reorder
...	expressions evaluated in the context of df and then fed to order

Examples

```
mtcars[with(mtcars, order(cyl, disp)), ]
arrange(mtcars, cyl, disp)
arrange(mtcars, cyl, desc(disp))
```

[Package *plyr* version 1.7.1 Index]

Figure 25.1: HTML rendering of arrange documentation

When you use `help` or `example` it looks for the Rd files in the *installed* package. This isn't very useful for package development, because we want to retrieve the `.Rd` files from the *source* package. `devtools` provides two functions to do this `dev_help` and `dev_example` - these behave similarly to `help` and `example`, but look in source packages you have loaded with `load_all`, rather than installed packages you've loaded with `library`.

Common documentation tags

As you've seen roxygen comments have a special format. They start with `#'` and they include tags of the form `@tagname` that break the documentation up into pieces. The content of a tag extends from the tag name to the next tag, and they can span multiple lines.

If you want to put a literal `@` in the documentation, use `@@`.

Each documentation block starts with a text description. The first sentence/line becomes the title of the documentation. That's what you see when you look at `help(package = mypackage)` and is shown at the top of the documentation for each package. It should be succinct. The second paragraph is the description: this is the first thing shown in the documentation and should briefly describe what the function does. The third and subsequence paragraphs go in the details: this is a (often long) section that comes after the description of the parameters. If you want to override the default behaviour, you can use the `@title`, `@description` and `@details` tags.

Here's an example showing what the documentation for `sum` might look like if it was written using roxygen:

```
#' Sum of Vector Elements
#'
#' \code{sum} returns the sum of all the values present in its arguments.
#'
#' This is a generic function: methods can be defined for it directly
#' or via the \code{Summary} group generic.  For this to work properly,
#' the arguments \code{...} should be unnamed, and dispatch is on the
#' first argument.'
sum <- function(..., na.rm = TRUE) {}
```

The following documentation is equivalent, but uses explicit tags. This is not necessary unless you want to have a multiple paragraph description or title, or want to omit the description (in which case roxygen will replace with the title).

```
#' @title Sum of Vector Elements
#'
#' @description
#' \code{sum} returns the sum of all the values present in its arguments.
#'
#' @details
#' This is a generic function: methods can be defined for it directly
#' or via the \code{Summary} group generic.  For this to work properly,
#' the arguments \code{...} should be unnamed, and dispatch is on the
#' first argument.'
sum <- function(..., na.rm = TRUE) {}
```

If you use explicit tags, you can put them in any order and still get the same output. The following documentation block would produce exactly the same Rd file and final documentation as the previous two.

```
#' @details
#' This is a generic function: methods can be defined for it directly
#' or via the \code{Summary} group generic.  For this to work properly,
#' the arguments \code{...} should be unnamed, and dispatch is on the
#' first argument.'
```

```
 #'
 #' @title Sum of Vector Elements
 #'
 #' @description
#' \code{sum} returns the sum of all the values present in its arguments.
 #'
 sum <- function(..., na.rm = TRUE) {}
```

You can also opt to document multiple functions in one `.Rd` file. If you want to do so, use the `@rdname` tag to manully specify the name of the rd file (if you omit this it's done automatically in a way that every object gets its own documentation). You should be careful about documenting too many functions in one file because it gets confusing (which parameter belongs to which function?), but it can be useful if the functions are tightly connected.

You can get documentation on a given tag with `?TagName`. For example, if you wanted to get help on the param tag, you'd do `?TagParam`. (Unfortunately you can't do `?@param` because of a technical limitation in Rd files.)

Documenting functions (and methods)

When documenting a function, the first decision you need to make is whether you want to export it or not. Exporting is described more in [[namespaces]], but basically when you export a function you are saying it is ready for use and you are making a commitment to keep it around in a similar form for the near future. All exported functions need to be documented. You don't have to document internal functions, but it can be a good idea if they're particularly complicated. Few users will read the documentation for internal functions, but it will be useful for you so when you come back to them in the future you don't need to struggle to recall what the inputs and outputs are.

The following code shows the arrange function and its documentation from the plyr package.

```
#' Order a data frame by its columns.
#'
#' This function completes the subsetting, transforming and ordering triad
#' with a function that works in a similar way to \code{\link{subset}} and
#' \code{\link{transform}} but for reordering a data frame by its columns.
#' This saves a lot of typing!
#'
#' @param df data frame to reorder
#' @param ... expressions evaluated in the context of \code{df} and
#'   then fed to \code{\link{order}}
#' @keywords manip
#' @export
#' @examples
#' mtcars[with(mtcars, order(cyl, disp)), ]
#' arrange(mtcars, cyl, disp)
#' arrange(mtcars, cyl, desc(disp))
arrange <- function(df, ...) {
  ord <- eval(substitute(order(...)), df, parent.frame())
  unrowname(df[ord, ])
}
```

The tags used here are:

- `@param arg description` - a description for each function argument. This can span multiple lines (or even paragraphs) if necessary.
- `@export` - a flag indicating that this function should be exported for use by others. Described in more detail in [[namespaces]].
- `@examples` - examples of the function in use. In my opinion, this is the most important part of the documentation - these are what most people will look at first to figure out how to use the function. I always put this last.

Other tags that you might find useful are:

- `@return` - the type of object that the function returns

The following tags apply to all types of documentation

- `@author`, in the form `Firstname Lastname <email@@address.com>`. Use this if some of the functions are written by people other than the author of the package.
- `@seealso` - to provide pointers to other related topics
- `@references` - references to scientific literature on this topic
- `@aliases` - a list of additional topic names that will be mapped to this documentation when the user looks them up from the command line.
- `@family` - a family name. All functions that have the same family tag will be linked in the documentation.

Documenting S3 generic functions and methods

S3 generics are documented in the same way as functions - there is typically no reason for a user to know (or care) that a function is a generic. (Although you may want to mention it in the docs so that other developers know that they can extend it with their own methods).

If you `@export` an S3 method, all methods defined in your package will also get exported appropriately (using the `S3method()` namespace directive): they will be available when you call the generic, but users can not call the methods directly. For example, the `wday` generic in `lubridate` is exported:

```
library(lubridate)
wday(Sys.Date(), label = TRUE)
wday(1, label = TRUE)
```

But you can't call any of the methods directly:

```
wday.default(Sys.Date(), label = TRUE)
wday.numeric(1, label = TRUE)

lubridate:::wday.default(Sys.Date(), label = TRUE)
lubridate:::wday.numeric(1, label = TRUE)
```

This is good practice because it hides the internal implementation details of your functions. Users should not rely on specific behaviour (for example, in this case we need to call `wday.default` for Date objects). This makes it easier for you to change how the functions work in the future - for example, you could change from S3 to S4 dispatch.

You only need to explicitly `@export` a method if it's for a generic defined in another package. This is one of the easiest things to forget, and it creates subtle bugs that `R CMD check` doesn't report and are hard to track down. Future versions of roxygen should hopefully make this easier.

Most of the time you don't need to document S3 methods - particularly if they are for simple generics like `print`. However, if your method is more complicated, you should document it. In base R, you can find documentation for more complex methods like `predict.lm`, `predict.glm`, `anova.glm` and so on.

Documenting S4 generic and methods

The same rules apply when documenting S4 generics and methods:

- Document the generic like you would document any function. Document methods that are more complex or have significant differences in behaviours.
- You don't need to export methods if they're for a generic you defined (just export the generic). Export all methods for generics defined in other packages.
- Use `@genericMethods` in the generic documentation if you want an automated listing of all methods implemented for the generic.

Documenting R5 methods

Currently there is no way to use roxygen to document R5 methods. Use the standard docstring format. Alternatively, use roxygen to document an object with `@name` named by convention and assign it to `NULL`. For example, if object `obj` has method `getName`, then document the method as follows, using the convention to replace the `$` with `_`:

```
#' @title Get the name of the object
#' ...
#' @name obj_getName
NULL

obj$methods(getName = function() return("name"))
```

And the help for `getName` can be accessed by `?obj_getName`

Documenting classes

Documenting a S3 class

Since S3 classes have no formal structure, you should document the constructor function.

Documenting an S4 class

Typically you should document the `setClass` call, and if present, the constructor function. Unless you want to separate out internal documentation for the class and the public interface (for the constructor), I recommend using `@rdname` to document both in the same file. This will mean that `?classname` and `class?classname` go to the same place.

Export the class if you want other developers to be able to write subclasses of your class. Export the constructor if you want users to be able to use it.

Documenting an R5 class

Documenting datasets

The following documentation excerpt comes from the diamonds dataset in ggplot2. We can't document the object directly (because it lives in the data directory), so we document `NULL` and provide the `@name` of the object we really want to document.

```
#' Prices of 50,000 round cut diamonds.
#'
#' A dataset containing the prices and other attributes of almost 54,000
#'  diamonds. The variables are as follows:
#'
#' \itemize{
#'   \item price. price in US dollars (\$326--\$18,823)
#'   \item carat. weight of the diamond (0.2--5.01)
#'   ...
#' }
#'
#' @docType data
#' @keywords datasets
#' @format A data frame with 53940 rows and 10 variables
#' @name diamonds
NULL
```

There are a few new tags:

- `@docType data`, which indicates that this is documentation for a dataset.
- `@format`, which gives an overview of the structure of the dataset. If you omit this, roxygen will automatically add something based on the first line of `str` output
- `@source` where you got the data form, often a `\url{}`.

Documenting packages

As well [[package level documentation|documenting-packages]] resources, every package should also have its own documentation page. I usually put this documentation in a file with the same name as the package.

This documentation topic should contain an overview documentation topic that describes the overall purpose of the package, and points to the most important functions. This topic should have `@docType package` and be aliased to `package-pkgname` and `pkgname` (unless there is already a function by that name) so that you can get an overview of the package by doing `?pkgname` or `package?pkgname`.

There are still relatively few packages that provide package documentation, but it's an extremely useful tool for users, because instead of just listing functions like `help(package = pkgname)` it organises them and shows the user where to get started.

The example below shows the basic structure, as taken from the documentation for the `lubridate` package:

```
#' Dates and times made easy with lubridate.
#'
#' Lubridate provides tools that make it easier to parse and
#' manipulate dates. These tools are grouped below by common
#' purpose. More information about each function can be found in
#' its help documentation.
#'
#' Parsing dates
#'
#' Lubridate's parsing functions read strings into R as POSIXct
#' date-time objects. Users should choose the function whose name
#' models the order in which the year ('y'), month ('m') and day
#' ('d') elements appear the string to be parsed:
#' \code{\link{dmy}}, \code{\link{myd}}, \code{\link{ymd}},
#' \code{\link{ydm}}, \code{\link{dym}}, \code{\link{mdy}},
#' \code{\link{ymd_hms}}).
#'
#' ...
#'
#' @references Garrett Grolemund, Hadley Wickham (2011). Dates and Times
#'  Made Easy with lubridate. Journal of Statistical Software, 40(3),
#'   1-25. \url{http://www.jstatsoft.org/v40/i03/}.
#' @import plyr stringr
#' @docType package
#' @name lubridate
NULL
```

Important components are:

- The general overview of the package: what it does, and what are the important pieces.

- Like for data sets, there isn't a object that we can document directly so document `NULL` and use `@name` to say what we're actually documenting
- `@docType package` to indicate that it's documenting a package. This will automatically add the corect aliases so that `package?yourpackage` works.
- `@references` point to any published material about the package that users might find help.

The package documentation should not contain a verbatim list of functions or copy of `DESCRIPTION`. This file is for human reading, so pick the most important elements of your package.

Text formatting

Within roxygen text, you use the usual R documentation formatting rules, as summarised below. A fuller description is available in the R extensions[5] manual.

Sections are indicated by the @section tag, followed by the title (which should be in sentence case) and a colon. Subsections are indicated by a LaTeX-style command; the first argument is the subsection title and the second argument contains the subsection content. The following example illustrates how to add an arbitrary section:

```
@section Warning:
  You must not call this function unless ...

  \subsection{Exceptions}{
     Apart from the following special cases...
  }
```

The section ends when another tag is declared or when the roxygen2 block ends.

Lists

- Ordered (numbered) lists:

  ```
  \enumerate{
    \item First item
    \item Second item
  }
  ```

- Unordered (bulleted) lists

[5] http://cran.r-project.org/doc/manuals/R-exts.html#Sectioning

```
\itemize{
  \item First item
  \item Second item
}
```

- Definition (named) lists

```
\describe{
  \item{One}{First item}
  \item{Two}{Second item}
}
```

Tables

Tables are created with the tabular command, which has two arguments:

- Column alignment: a letter for each column. `l` = left alignment, `c` = centre, `r` = right.
- Table contents. Columns separated by `\tab`, rows separated by `\cr`.

The following function will turn an R data frame into the correct format. It ignores column and row names, but should get you started.

```
tabular <- function(df) {
  stopifnot(is.data.frame(df))
  align <- function(x) if (is.numeric(x)) "r" else "l"
  col_align <- vapply(df, align, character(1))

  contents <- do.call("paste", c(lapply(df, format),
    list(sep = " \\tab ", collapse = "\\cr\n")))

  mat <- matrix(unlist(lapply(df, format)), ncol = ncol(df))
  paste("\\tabular{", paste(col_align, collapse = ""), "}{\n",
    contents, "}\n", sep = "")
}
```

Mathematics

`\eqn` for inline, `\deqn` for display. Standard LaTeX (no extensions).

Character formatting

- `\emph{text}`: emphasised text, usually displayed as *italics*
- `\strong{text}`: strong text, usually displayed in **bold**
- `\code{text}`, `\pkg{package_name}`, `\file{file_name}`
- External links: `\email{email_address}`, `\url{url}`, `\href{url}{text}`
- `\link[package]{function}` - the first argument can be omitted if the link is in the current package, or the base package. Will usually be wrapped inside `\code`: `\code{\link{fun}}`

Dynamic help

Since R.XX, help has been…

Chapter 26

Testing

Testing is important to make sure that your package behaves as you expect it to do. You probably test your code already, but you may not have taken the next step to automate it. This chapter describes how to use the `testthat` package to create automated tests for your code.

Motivation

I started automating my tests because I discovered I was spending too much time recreating bugs that I had previously fixed. While I was writing the original code or fixing the bug, I'd perform many interactive tests to make sure the code worked, but I never had a system for retaining these tests and running them, again and again. I think that this is a common development practice of R programmers: it's not that we don't test our code, it's that we don't store our tests so they can be re-run automatically.

It will always require a little more work to turn your casual interactive tests into reproducible scripts: you can no longer visually inspect the output, so instead you have to write code that does the inspection for you. However, this is an investment in the future of your code that will pay off in:

- Decreased frustration. Whenever I'm working to a strict deadline I always seem to discover a bug in old code. Having to stop what I'm doing to fix the bug is a real pain. This happens less when I do more testing, and I can easily see which parts of my code I can be confident in by looking at how well they are tested.
- Better code structure. Code that's easy to test is usually better designed. I have found writing tests makes me extract out the complicated parts of my code into separate functions that work in isolation. These functions

are easier to test, have less duplication, are easier to understand and are easier to re-combine in new ways.

- Less struggle to pick up development after a break. If you always finish a session of coding by creating a failing test (e.g. for the feature you want to implement next) it's easy to pick up where you left off: your tests let you know what to do next.
- Increased confidence when making changes. If you know that all major functionality has a test associated with it, you can confidently make big changes with out worrying about accidentally breaking something. For me, this is particularly useful when I think of a simpler way to accomplish a task - often my simpler solution is only simpler because I've forgotten an important use case!

testthat test structure

`testthat` has a hierarchical structure made up of expectations, tests and contexts.

- An **expectation** describes the expected result of a computation: Does it have the right value and right class? Does it produce error messages when it should? There are 11 types of built in expectations.
- A **test** groups together multiple expectations to test one function, or tightly related functionality across multiple functions. A test is created with the `test_that` function.
- A **context** groups together multiple tests that test related functionality. Contexts are defined with the `context()` function.

These are described in detail below.

Expectations give you the tools to convert your visual, interactive experiments into reproducible scripts; tests and contexts are ways of organising your expectations so that when something goes wrong you can easily track down the source of the problem.

Expectations

An expectation is the finest level of testing; it makes a binary assertion about whether or not a value is as you expect. If the expectation isn't true, `testthat` will raise an error.

An expectation is easy to read, since it is nearly a sentence already: `expect_that(a, equals(b))` reads as "I expect that a will equal b".

There are 11 built in expectations:

- `equals()` uses `all.equal()` to check for equality with numerical tolerance.

  # Passes expect_that(10, equals(10)) # Also passes expect_that(10, equals(10 + 1e-7))
  # Fails expect_that(10, equals(10 + 1e-6))
  # Definitely fails! expect_that(10, equals(11))

- `is_identical_to()` uses `identical()` to check for exact equality.

  # Passes expect_that(10, is_identical_to(10)) # Fails expect_that(10, is_identical_to(10 + 1e-10))

- `is_equivalent_to()` is a more relaxed version of `equals()` that ignores attributes:

  # Fails expect_that(c("one" = 1, "two" = 2), equals(1:2)) # Passes expect_that(c("one" = 1, "two" = 2), is_equivalent_to(1:2))

- `is_a()` checks that an object `inherit()`s from a specified class.

  model <- lm(mpg ~ wt, data = mtcars) # Passes expect_that(model, is_a("lm"))
  # Fails expect_that(model, is_a("glm"))

- `matches()` matches a character vector against a regular expression. The optional `all` argument controls whether all elements or just one element need to match. This code is powered by `str_detect()` from the `stringr` package.

  string <- "Testing is fun!" # Passes expect_that(string, matches("Testing"))
  # Fails, match is case-sensitive expect_that(string, matches("testing"))
  # Passes, match can be a regular expression expect_that(string, matches("t.+ting"))

- `prints_text()` matches the printed output from an expression against a regular expression.

  a <- list(1:10, letters) # Passes expect_that(str(a), prints_text("List of 2")) # Passes expect_that(str(a), prints_text(fixed("int [1:10]"))

- `shows_message()` checks that an expression shows a message:

  # Passes expect_that(library(mgcv), shows_message("This is mgcv"))

- `gives_warning()` expects that you get a warning.

  # Passes expect_that(log(-1), gives_warning()) expect_that(log(-1), gives_warning("NaNs produced")) # Fails expect_that(log(0), gives_warning())

- `throws_error()` verifies that the expression throws an error. You can also supply a regular expression which is applied to the text of the error.

  # Fails expect_that(1 / 2, throws_error()) # Passes expect_that(1 / “a”, throws_error()) # But better to be explicit expect_that(1 / “a”, throws_error(“non-numeric argument”))

- `is_true()` is a useful catchall if none of the other expectations do what you want - it checks that an expression is true. `is_false()` is the complement of `is_true()`.

If you don’t like the readable, but verbose, `expect_that` style, you can use one of the shortcut functions:

Full

Abbreviation

expect_that(x, is_true())

expect_true(x)

expect_that(x, is_false())

expect_false(x)

expect_that(x, is_a(y))

expect_is(x, y)

expect_that(x, equals(y))

expect_equal(x, y)

expect_that(x, is_equivalent_to(y))

expect_equivalent(x, y)

expect_that(x, is_identical_to(y))

expect_identical(x, y)

expect_that(x, matches(y))

expect_match(x, y)

expect_that(x, prints_text(y))

expect_output(x, y)

expect_that(x, shows_message(y))

expect_message(x, y)

expect_that(x, gives_warning(y))

expect_warning(x, y)

expect_that(x, throws_error(y))

expect_error(x, y)

Running a sequence of expectations is useful because it ensures that your code behaves as expected. You could even use an expectation within a function to check that the inputs are what you expect. However, they're not so useful when something goes wrong: all you know is that something is not as expected, not anything about where the problem is. Tests, described next, organise expectations into coherent blocks that describe the overall goal of a set of expectations.

Tests

Each test should test a single item of functionality and have an informative name. The idea is that when a test fails, you should know exactly where to look for the problem in your code. You create a new test with `test_that`, with parameters name and code block. The test name should complete the sentence "Test that" and the code block should be a collection of expectations. When there's a failure, it's the test name that will help you figure out what's gone wrong.

The following code shows one test of the `floor_date` function from `lubridate`. There are 7 expectations that check the results of rounding a date down to the nearest second, minute, hour, etc. Note how we've defined a couple of helper functions to make the test more concise so you can easily see what changes in each expectation.

```
test_that("floor_date works for different units", {
  base <- as.POSIXct("2009-08-03 12:01:59.23", tz = "UTC")

  is_time <- function(x) equals(as.POSIXct(x, tz = "UTC"))
  floor_base <- function(unit) floor_date(base, unit)

  expect_that(floor_base("second"), is_time("2009-08-03 12:01:59"))
  expect_that(floor_base("minute"), is_time("2009-08-03 12:01:00"))
  expect_that(floor_base("hour"),   is_time("2009-08-03 12:00:00"))
  expect_that(floor_base("day"),    is_time("2009-08-03 00:00:00"))
  expect_that(floor_base("week"),   is_time("2009-08-02 00:00:00"))
  expect_that(floor_base("month"),  is_time("2009-08-01 00:00:00"))
  expect_that(floor_base("year"),   is_time("2009-01-01 00:00:00"))
})
```

Each test is run in its own environment so it is self-contained. The exceptions are actions which have effects outside the local environment. These include things that affect:

- the filesystem: creating and deleting files, changing the working directory, etc.
- the search path: package loading & detaching, `{attach}`.
- `global options, like options() and par()`.

When you use these actions in tests, you'll need to clean up after yourself. Many other testing packages have set-up and teardown methods that are run automatically before and after each test. These are not so important with `testthat` because you can create objects outside of the tests and rely on R's copy-on-modify semantics to keep them unchanged between test runs. To clean up other actions you can use regular R functions.

You can run a set of tests just by `source()`ing a file, but as you write more and more tests, you'll probably want a little more infrastructure. The first part of that infrastructure is contexts, described below, which give a convenient way to label each file, helping to locate failures when you have many tests.

Contexts

Contexts group tests together into blocks that test related functionality, and are established with the code `context("My context")`. Normally there is one context per file, but you can have more if you want, or you can use the same context in multiple files.

The following code shows the context that tests the operation of the `str_length()` function in `stringr`. The tests are very simple, but cover two situations where `nchar()` in base R gives surprising results.

```
context("String length")

test_that("str_length is number of characters", {
  expect_that(str_length("a"), equals(1))
  expect_that(str_length("ab"), equals(2))
  expect_that(str_length("abc"), equals(3))
})

test_that("str_length of missing is missing", {
  expect_that(str_length(NA), equals(NA_integer_))
  expect_that(str_length(c(NA, 1)), equals(c(NA, 1)))
  expect_that(str_length("NA"), equals(2))
})

test_that("str_length of factor is length of level", {
  expect_that(str_length(factor("a")), equals(1))
  expect_that(str_length(factor("ab")), equals(2))
  expect_that(str_length(factor("abc")), equals(3))
})
```

Running tests

There are two situations in which you want to run your tests: interactively while you're developing your package to make sure that everything works ok, and then as a final automated check before releasing your package.

- run all tests in a file or directory `test_file()` or `test_dir()`
- automatically run tests whenever something changes with `autotest`.
- have `R CMD check` run your tests.

Testing files and directories

You can run all tests in a file with `test_file(path)`. The following code shows the difference between `test_file` and `source` for the `stringr` tests, as well as those same tests for `nchar`. You can see the advantage of `test_file` over `source`: instead of seeing the first failure, you see the performance of all tests.

```
> source("test-str_length.r")
> test_file("test-str_length.r")
.........

> source("test-nchar.r")
Error: Test failure in 'nchar of missing is missing'
* nchar(NA) not equal to NA_integer_
'is.NA' value mismatch: 0 in current 1 in target
* nchar(c(NA, 1)) not equal to c(NA, 1)
'is.NA' value mismatch: 0 in current 1 in target

> test_file("test-nchar.r")
...12..34

1. Failure: nchar of missing is missing ---------------------------------
nchar(NA) not equal to NA_integer_
'is.NA' value mismatch: 0 in current 1 in target

2. Failure: nchar of missing is missing ---------------------------------
```

```
nchar(c(NA, 1)) not equal to c(NA, 1)
'is.NA' value mismatch: 0 in current 1 in target

3. Failure: nchar of factor is length of level --------------------------
nchar(factor("ab")) not equal to 2
Mean relative difference: 0.5

4. Failure: nchar of factor is length of level --------------------------
nchar(factor("abc")) not equal to 3
Mean relative difference: 0.6666667
```

Each expectation is displayed as either a green dot (indicating success) or a red number (including failure). That number indexes into a list of further details, printed after all tests have been run. What you can't see is that this display is dynamic: new dot is printed each time a test passes, and it's rather satisfying to watch.

`test_dir` will run all of the test files in a directory, assuming that test files start with `test` (so it's possible to intermix regular code and tests in the same directory). This is handy if you're developing a small set of scripts rather than a complete package. The following shows the output from the `stringr` tests. You can see there are 12 contexts with between 2 and 25 expectations each. As you'd hope in a released package, all the tests pass.

```
> test_dir("inst/tests/")
String and pattern checks : ......
Detecting patterns : .........
Duplicating strings : ......
Extract patterns : ..
Joining strings : ......
String length : .........
Locations : ............
Matching groups : ..............
Test padding : ....
Splitting strings : .........................
Extracting substrings : ...................
Trimming strings : ........
```

If you want a more minimal report, suitable for display on a dashboard, you can use a different reporter. `testthat` comes with three reporters: stop, minimal and summary. The stop reporter is the default and `stop()`s whenever a failure is encountered, and the summary report is the default for `test_file` and `test_dir`. The minimal reporter is shown below: it prints `.` for success, `E` for an error and `F` a failure. The following output shows the results of running the `stringr` test suite with the minimal reporter.

```
> test_dir("inst/tests/", reporter="minimal")
..............................................
```

Autotest

Tests are most useful when run frequently, and `autotest` takes that idea to the limit by rerunning your tests whenever your code or tests changes. `autotest()` has two arguments, `code_path` and `test_path`, which point to a directory of source code and tests respectively.

Once run, `autotest()` will continuously scan both directories for changes. If a test file is modified, it will test that file; if a code file is modified, it will reload that file and rerun all tests. To quit, you'll need to press Ctrl + Break on Windows, Escape in the Mac gui, or Ctrl + C if running from the command line.

This promotes a workflow where the *only* way you test your code is through tests. Instead of modify-save-source-check you just modify and save, then watch the automated test output for problems.

R CMD check

When developing a package, put your tests in `inst/tests` (note that your test files must begin with `test`), and then create a file `tests/run-all.R` (note that it must be a capital R), which contains the following code:

```
library(testthat)
library(mypackage)

test_package("mypackage")
```

This will evaluate your tests in the package namespace (so you can test non-exported functions), and it will throw an error if there are any test failures. This means you'll see the full report of test failures and `R CMD check` won't pass unless all tests pass.

This also makes it easy for your users to check that you package works correctly in their run-time environment.

Chapter 27

Style guide

Good coding style is like using correct punctuation when writing: you can manage without it, but it sure makes things easier to read. As with punctuation, there are many possible variations, and the main thing is to be consistent. The following guide describes the style that I use - you don't have to use it, but you need to have some consistent style that you do follow. My style is based on Google's R style guide[1], with a few tweaks.

Good style is important because while your code only has one author, it will usually have multiple readers, and when you know you will be working with multiple people on the same code, it's a good idea to agree on a common style up-front. No style is uniformly better than any other style, and if you're working with a group of people, you may need to sacrifice some of your most favourite types of style.

One package that can make adhering to a style guide easier is `formatR`, by Yihui Xie. It can't do everything, but if you're starting with very poorly formatted R code, it will get you to a good place much more quickly than doing everything by hand. Make sure to read the notes on the wiki[2] before using it.

Notation and naming

File names

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

```
# Good
```

[1]http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html
[2]https://github.com/yihui/formatR/wiki

```
explore-diamonds.r
hadley-wickham-hw-1.r
# Bad
foo.r
my-homework.R
```

Identifiers

“There are only two hard things in Computer Science: cache invalidation and naming things.” – Phil Karlton

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

```
# Good
day_one
day_1
# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

Syntax

Spacing

Place spaces around all infix operators (`=`, `+`, `-`, `<-`, etc.). Do not place a space before a comma, but always place one after a comma (just like in regular English).

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

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

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

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

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

```
list(
  x = call_this_long_function(a, b),
  y = a * e / d ^ f)

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

Do not place spaces around code in parentheses or square brackets. (Except if there's a trailing comma: always place a space after a comma, just like in ordinary English.)

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

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

Curly braces

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

Always indent the code inside the curly braces.

```
# Good

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

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

# Bad

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

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

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

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

Line length

Keep your lines less than 80 characters. This is the amount that will fit comfortably on a printed page at a reasonable size. If you find you are running out of room, this is probably an indication that you should encapsulate some of the work in a separate function.

Indentation

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

The only exception is if a function definition runs over multiple lines: indent the second line to line up with where the definition starts:

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

Assignment

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

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

Organisation

Commenting guidelines

Comment your code. Entire commented lines should begin with `#` and one space. Comments should explain the why, not the what.

Use commented lines of `-` and `=` to break up your files into scannable chunks.

Chapter 28

Namespaces

As the name suggest, namespaces provide “spaces” for “names”, providing a context for evaluating which object is found when you look for it. When developing code, they allow you to specify which package to look for for a function when there are multiple packages it could come from, and when developing packages they ensure that your functions always call functions in the same place.

Namespaces make it possible for packages to refer to specific functions in other packages, not the functions that you have defined in the global workspace.

For example, take the simple `nrow` function:

```
nrow
# function (x)
# dim(x)[1L]
```

What happens if we create our own dim method? Does `dim` break?

```
dim <- function(x) c(1, 1)
dim(mtcars)
nrow(mtcars)
```

Suprisingly, it does not! That’s because when `nrow` looks for an object called `dim`, it finds the function in the base package, not our function.

Namespaces also provide an extension to the standard way of looking up the value of an object: instead of just typing `objectname`, you type `package::objectname`.

Namespaces also control the functions and methods that your package makes available for use by others. Namespaces make it easier to come up with your own function names without worrying about what names other packages have used. A namespace means you can use any name you like for internal functions, and when there is a conflict with an exported function, there is a standard disambiguation procedure.

The easiest way to use namespaces is with roxygen2, because it keeps the namespace definitions next to the function that it concerns. The translation between roxygen2 tags and `NAMESPACE` directives is usually straightforward: the tag becomes a function name and its space-separated arguments become comma separated arguments to the function. For example `@import plyr` becomes `import(plyr)` and `@importFrom plyr ddply` becomes `importFrom(plyr, ddply)`. The chief exception is `@export` which will automatically figure out the function name to export.

How do namespaces work

Exporting

For every function in your package, you need to decide whether it is external and available to all users of the package, or internal and only available to other functions within the package. It's not always easy to tell whether or not a function is internal or external. A few rules of thumb:

- Is the purpose of the function the same as the purpose of the package? If not, make it internal. (A package should provide a set of closely related functions for a well-defined problem domain - someone should be able to look at all the functions in your package and say this is a package about X - if not, you should consider splitting it up into two packages)
- Does the function have a clear purpose and can you easily explain it? Every external function needs to be documented, and there's an implicit contract that that function will continue to exist in the future. If that's not the case, don't export it.

If a function isn't exported, you don't need to document it. This doesn't mean you shouldn't document it, but you only need to if it's complicated enough that you think you won't remember what it does. Generally, you want to export as few functions as possible: this makes it easier to change the package in the future.

As described below, the `@export` tag is all that you need. There are only a few exceptions:

- **Functions**: use the `@export` tag.

- **S3 methods**: There are two ways to export a method for an S3 method depending on whether it's documented or not:
  - If it's documented, you'll already be using the `@method` tag to state that it's an S3 method and you only need the `@export` to generate the correct export flag in the `NAMESPACE`
  - If it's not documented, use the `S3method` tag: `@S3method function class`
- **S4 classes**: Use `@export`
- **S4 methods**: If the methods are for classes that you have defined and exported, you don't need to do anything. If they are for classes defined in other packages, you need to use `@export`.
- **Other objects**: For any other types of object that you want to make available to the user, use `@export`.

You may also want to make the distinction between functions for users and functions for other developers. Functions that might be useful for developers or power users should be exported, but tagged with `@keywords internal` so they don't show up in routine lists of function documentation.

Importing

In your package `DESCRIPTION` there are two ways to indicate that your package requires another package to work: by listing it in either `Depends` or `Imports`. `Depends` works just like using library to load a package, but `Imports` is a little more subtle: the dependency doesn't get loaded in a way the user can see. This is good practice because it reduces the chances of conflict, and it makes the code clearer by requiring that every package used be explicitly loaded. Since R 2.14 there is no reason to use `Depends` because all packages have a namespace.

There are two places you need to record your package's dependency:

- In the `Imports` field in the `DESCRIPTION` file, used by `install.packages` to download package dependencies automatically.
- In the `NAMESPACE` file, to make all the functions in the dependency available to your code. The easiest way to do this is to add `@import package-name` to your package documentation[1]:

  ```
  #' @docType package
  #' ...
  #' @import stringr MASS
  ```

[1] Documenting-packages.html

and have `roxygen2` generate the `NAMESPACE` file from that.

There are two alternatives to using `@import`:

- Use `@importFrom package fun1 fun2 ...` to only import selected functions from another package. This is important if you are importing two packages share functions with the same name. You can also use it to produce a very specific `NAMESPACE`, but at the cost of having to use `@importFrom` for every function that uses a function from another package.
- `::` refers to a function within a package directly. I don't recommend this method because it doesn't work well during package development – it will use the installed version of the package, rather than the development version.
- S4 methods: See the R extensions[2] manual

You should very very very rarely use `:::`. This is a sign that you're using an internal function from someone else - and there is no guarantee that that function won't change from version to version. It would be better to encourage the author to make it an external, exported function, or ask if you could include a copy of it in your package.

Compiled code

If you have C or Fortran code in your package, you'll need to add `@useDynLib mypackage` to your package documentation to ensure your functions can access it. This means you don't need to specify `PACKAGE` in `.Call`.

How do they work

New set of rules on top of ordinary [[scoping]] rules, which deal with lists of environments - each environment belongs to a package. Search path. Variable look up differs depending on whether you're inside or outside a package. If a package has a namespace, then R looks first inside the package namespace, then the imports, then the base namespace and then the normal search path.

[2] http://cran.r-project.org/doc/manuals/R-exts.html#Name-spaces-with-S4-classes-and-methods

Chapter 29

Git and github

Git

Using a source code control system, like git, is highly recommended because it makes it easy to:

- incorporate contributions from multiple developers working on the code at the same time
- rewind time to undo mistakes or see what has changed between working code and broken code

In this document, I'll describe the use of git and github because they are the tools that I am most familiar with. There are many others (like subversion, mercurial and bazaar) that offer similar capabilities - the choice of git is somewhat arbitrary but the skills will readily transfer to other systems.

I'll just give you the basics, and give pointers to places were you can learn more advanced techniques.

http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html

The basics

- `git init`
- `git add`
- `git commit`
- `git push`
- `git pull --rebase`

Github

- git enabled wiki (which these documents have been written in)
- makes it easy for others to send in patches
- ticketing system for tracking bugs
- RSS feed of changes, and line-by-line comments are very useful for working with collaborators

Additionally github gives you free hosting for public repositories. You only have to pay if you want private repositories with private collaborators.

Chapter 30

Releasing a package

Checking

- from within R, run `roxygenise()`, or `devtools::document()` to update documentation
- from the command line, run `R CMD check`

Passing `R CMD check` is the most frustrating part of package development, and it usually takes some time the first time. Hopefully by following the tips elsewhere in this document you'll be in a good place to start – in particular, using roxygen and only exporting the minimal number of functions is likely to save a lot of work.

One place that it is frustrating to have problems with is the examples. If you discover a mistake, you need to fix it in the roxygen comments, rerun roxygen and then rerun `R CMD check`. The examples are one of the last things checked, so this process can be very time consuming, particularly if you have more than one bug. The `devtools` package contains a function, `run_examples` designed to make this somewhat less painful: all it does is run functions. It also has an optional parameter which tells it which function to start at - that way once you've discovered an error, you can rerun from just that file, not all the files that lead up to.

Version numbers

The version number of your package increases with subsequent releases of a package, but it's more than just an incrementing counter – the way the number changes with each release can convey information about what kind of changes are in the package.

An R package version can consist of a series of numbers, each separated with "." or "-". For example, a package might have a version 1.9. This version number is considered by R to be the same as 1.9.0, less than version 1.9.2, and all of these are less than version 1.10 (which is version "one point ten", not "one point one zero). R uses version numbers to determine whether package dependencies are satisfied. A package might, for example, import package `devtools (>= 1.9.2)`, in which case version 1.9 or 1.9.0 wouldn't work.

The version numbering advice here is inspired in part by Semantic Versionsing[1] and by the X.Org[2] versioning schemes.

A version number consists of up to three numbers, ... For version number 1.9.2, 1 is the major number, 9 is the minor number, and 2 is the patch number. If your version number is 2.0, then implicit patch number is 0.

As your package evolves, the way the version number changes can reflect the type of changes in the code:

- The major number changes when there are incompatible API changes.
- The minor number changes when there are backward-compatible API changes.
- The patch number changes with backwards-compatible fixes.
- Additionally, during development between released versions, the package has has a sub-patch version number of 99, as in 1.9.0.99. This makes it clear for users that they're using a development version of the package, as opposed to a formally released version.

Remember that these are guidelines. In practice, you might make changes that fall between the cracks. For example, if you make an API-incompatible change to a rarely-used part of your code, it may not deserve a major number change.

Publishing on CRAN

Once you have passed the checking process, you need to upload your package to CRAN. The checks will be run again with the latest development version of R, and on all platforms that R supports - this means that you should be prepare for more bugs to crop up. Don't get excited too soon!

- update `NEWS`, checking that dates are correct. Use `devtools::show_news` to check that it's in the correct format.

[1] http://semver.org
[2] http://www.x.org/releases/X11R7.7/doc/xorg-docs/Versions.html

- `R CMD build` then upload to CRAN: `ftp -u ftp://cran.R-project.org/incoming/package_name.tar.gz`
- send an email to `cran@r-project.org`, using the email address listed in the DESCRIPTION file. An example email would be something like: Hello, I just uploaded package name to CRAN. Please let me know if anything goes wrong. Thank you, Me. The subject line should be `CRAN submission PACKAGE VERSION`, this helps the CRAN maintainers keep track of the different submissions.

Once all the checks have passed you'll get a friendly email from the CRAN maintainer and you'll be ready to start publicising your package.

Publicising

Once you've received confirmation that all checks have passed on all platforms, you have a couple of technical operations to do:

- `git tag`, so you can mark exactly what version of the code this release corresponds to
- bump version in `DESCRIPTION` and `NEWS` files

Then you need to publicise your package. This is vitally important - for you hard work to be useful to someone, they need to know that it exists!

- send release announcement to `r-packages@stat.math.ethz.ch`. A release announcement should consist of a general introduction to your package (i.e. why should people care that you released a new version), and as well as what's new. I usually make these announcements by pasting together the package `README` and the appropriate section from the `NEWS`.
- announce on twitter, blog etc.
- Finally, don't forget to update your package webpage. If you don't have a package webpage – create one! There you can announce new versions, point to help resources, videos and talks about the package. If you're using github, I'd recommend using github pages[3] to create the website.

[3]http://pages.github.com/