R packages

Hadley Wickham

June 29, 2015

Contents

Part I

Getting started

Chapter 1

Introduction

In R, the fundamental unit of shareable code is the package. A package bundles together code, data, documentation, and tests, and is easy to share with others. As of January 2015, there were over 6,000 packages available on the **C**omprehensive **R** **A**rchive **N**etwork, or CRAN, the public clearing house for R packages. This huge variety of packages is one of the reasons that R is so successful: the chances are that someone has already solved a problem that you're working on, and you can benefit from their work by downloading their package.

If you're reading this book, you already know how to use packages:

- You install them from CRAN with `install.packages("x")`.
- You use them in R with `library("x")`.
- You get help on them with `package?x` and `help(package = "x")`.

The goal of this book is to teach you how to develop packages so that you can write your own, not just use other people's. Why write a package? One compelling reason is that you have code that you want to share with others. Bundling your code into a package makes it easy for other people to use it, because like you, they already know how to use packages. If your code is in a package, any R user can easily download it, install it and learn how to use it.

But packages are useful even if you never share your code. As Hilary Parker says in her introduction to packages (`http://hilaryparker.com/2014/04/29/writing-an-r-package-from-scratch/`): "Seriously, it doesn't have to be about sharing your code (although that is an added benefit!). It is about saving yourself time." Organising code in a package makes your life easier because packages come with conventions. For example, you put R code in `R/`, you put tests in `tests/` and you put data in `data/`. These conventions are helpful because:

- They save you time — you don't need to think about the best way to organise a project, you can just follow a template.

- Standardised conventions lead to standardised tools — if you buy into R's package conventions, you get many tools for free.

It's even possible to use packages to structure your data analyses, as Robert M Flight discusses in a series of blog posts (`http://rmflight.github.io/posts/2014/07/analyses_as_packages.html`).

1.1 Philosophy

This book espouses my philosophy of package development: anything that can be automated, should be automated. Do as little as possible by hand. Do as much as possible with functions. The goal is to spend your time thinking about what you want your package to do rather than thinking about the minutiae of package structure.

This philosophy is realised primarily through the devtools package, a suite of R functions that I wrote to automate common development tasks. The goal of devtools is to make package development as painless as possible. It does this by encapsulating all of the best practices of package development that I've learned over the years. Devtools protects you from many potential mistakes, so you can focus on the problem you're interested in, not on developing a package.

Devtools works hand-in-hand with RStudio, which I believe is the best development environment for most R users. The only real competitor is ESS (`http://ess.r-project.org/`), emacs speaks statistics, which is a rewarding environment if you're willing to put in the time to learn emacs and customise it to your needs. The history of ESS stretches back over 20 years (predating R!), but it's still actively developed and many of the workflows described in this book are also available there.

Together, devtools and RStudio insulate you from the low-level details of how packages are built. As you start to develop more packages, I highly recommend that you learn more about those details. The best resource for the official details of package development is always the official writing R extensions (`http://cran.r-project.org/doc/manuals/R-exts.html#Creating-R-packages`) manual. However, this manual can be hard to understand 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 book does. Writing R extensions is a useful resource once you've mastered the basics and want to learn what's going on under the hood.

1.2 In this book

You'll start by learning about the basic structure of a package, and the forms it can take, in Chapter 2. Then each of the next ten chapters of the book goes into more details about each component. They're roughly organised in order of importance:

- Chapter 3: the most important directory is `R/`, where your R code lives. A package with just this directory is still a useful package. (And indeed, if you stop reading the book after this chapter, you'll have still learned some useful new skills.)

- Chapter 4: the `DESCRIPTION` lets you describe what your package needs to work. If you're sharing your package, you'll also use the `DESCRIPTION` to describe what it does, who can use it (the license) and who to contact if things go wrong.

- Chapter 5: if you want other people (including future-you!) to understand how to use the functions in your package, you'll need to document them. I'll show you how to use roxygen2 to document your functions. I recommend roxygen2 because it lets you write code and documentation together while continuing to produce R's standard documentation format.

- Chapter 6: function documentation describes the nit-picky details of every function in your package. Vignettes give the big picture. They're long-form documents that show how to combine multiple parts of your package to solve real problems. I'll show you how to use Rmarkdown and knitr to create vignettes with a minimum of fuss.

- Chapter 7: to ensure your package works as designed (and continues to work as you make changes), it's essential to write unit tests which define correct behaviour, and alert you when functions break. In this chapter, I'll teach you how to use the testthat package to convert the informal interactive tests that you're already doing to formal, automated tests.

- Chapter 8: to play nicely with others, your package needs to define what functions it makes available to other packages and what functions it requires from other packages. This is the job of the `NAMESPACE` file and I'll show you how to use roxygen2 to generate it for you. The `NAMESPACE` is one of the more challenging parts of developing an R package but it's critical to master if you want your package to work reliably.

- Chapter 9: the `data/` directory allows you to include data with your package. You might do this to bundle data in a way that's easy for R users to access, or just to provide compelling examples in your documentation.

- Chapter 10: R code is designed for human efficiency, not computer efficiency, so it's useful to have a tool in your back pocket that allows you to write fast code. The `src/` directory allows you to include speedy compiled C and C++ code to solve performance bottlenecks in your package.

- Chapter 12: this chapter documents the handful of other components that are rarely needed: `demo/`, `exec/`, `po/` and `tools/`.

The final three chapters describe general best practices not specifically tied to one directory:

- Chapter 13: mastering a version control system is vital to easily collaborate with others, and is useful even for solo work because it allows you to easily undo mistakes. In this chapter, you'll learn how to use the popular git and GitHub combo with RStudio.
- Chapter 14: R provides very useful automated quality checks in the form of `R CMD check`. Running them regularly is a great way to avoid many common mistakes. The results can sometimes be a bit cryptic, so I provide a comprehensive cheatsheet to help you convert warnings to actionable insight.
- Chapter 15: the life-cycle of a package culminates with release to the public. This chapter compares the two main options (CRAN and github) and offers general advice on managing the process.

This is a lot to learn, but don't feel overwhelmed. Start with a minimal subset of useful features (e.g. just an `R/` directory!) and build up over time. To paraphrase the Zen monk Shunryu Suzuki: "Each package is perfect the way it is — and it can use a little improvement".

1.3 Getting started

To get started, make sure you have the latest version of R (at least 3.2.0, which is the version that the code in this book uses), then run the following code to get the packages you'll need:

```
install.packages(c("devtools", "roxygen2", "testthat", "knitr"))
```

Make sure you have a recent version of RStudio. You can check you have the right version by running:

```
install.packages("rstudioapi")
rstudioapi::isAvailable("0.99.149")
```

If not, you may need to install the preview version from `http://www.rstudio.com/products/rstudio/download/preview/`. This gives you access to the latest and greatest features, and only slightly increases your chances of finding a bug.

Use the following code to access new devtools functions as I develop them. This is particularly important during the development of the book.

```
devtools::install_github("hadley/devtools")
```

You'll also need a C compiler and a few other command line tools. If you're on Windows or Mac and you don't already have them, RStudio will install them for you. Otherwise:

- On Windows, download and install Rtools (http://cran.r-project.org/bin/windows/Rtools/). NB: this is not an R package!
- On Mac, make sure you have either XCode (available for free in the App Store) or the "Command Line Tools for Xcode" (http://developer.apple.com/downloads). You'll need to have a (free) Apple ID.
- On Linux, make sure you've installed not only R, but also the R development tools. For example, on Ubuntu (and Debian) you need to install the r-base-dev package.

You can check that you have everything installed and working by running the following code:

```
library(devtools)
has_devel()
#> '/Library/Frameworks/R.framework/Resources/bin/R' --vanilla CMD SHLIB foo.c
#>
#> clang -I/Library/Frameworks/R.framework/Resources/include -DNDEBUG
#>   -I/usr/local/include -I/usr/local/include/freetype2 -I/opt/X11/include
#>   -fPIC  -Wall -mtune=core2 -g -O2  -c foo.c -o foo.o
#> clang -dynamiclib -Wl,-headerpad_max_install_names -undefined dynamic_lookup
#>   -single_module -multiply_defined suppress -L/usr/local/lib -o foo.so foo.o
#>   -F/Library/Frameworks/R.framework/.. -framework R -Wl,-framework
#>   -Wl,CoreFoundation
[1] TRUE
```

This will print out some code which I use to help diagnose problems. If everything is ok, it will return TRUE. Otherwise, it will throw an error and you'll need to investigate what the problem is.

1.4 Acknowledgments

The tools in this book wouldn't be possible without many open source contributors. Winston Chang (https://github.com/wch/), my co-author on devtools, spent hours debugging painful S4 and compiled code problems so that devtools can quickly reload code for the vast majority of packages. Kirill Müller (https:

//github.com/krlmlr) contributed great patches to many of my package development packages including devtools, testthat, and roxygen2. Kevin Ushey (http://github.com/kevinushey), JJ Allaire (https://github.com/jjallaire) and Dirk Eddelbuettel (http://dirk.eddelbuettel.com) tirelessly answered all my basic C, C++ and Rcpp questions. Peter Danenburg (https://github.com/klutometis) and Manuel Eugster (http://www.statistik.lmu.de/~eugster/) wrote the first version of roxygen2 during a Google Summer of Code. Craig Citro (https://github.com/craigcitro) wrote much of the code to allow travis to work with R packages.

Often the only way I learn how to do it the right way is by doing it the wrong way first. For suffering many of package development errors, I'd like to thank all the CRAN maintainers, especially Brian Ripley, Uwe Ligges and Kurt Hornik.

This book was written in the open (https://github.com/hadley/r-pkgs/) and it is truly a community effort: many people read drafts, fixed typos, suggested improvements, and contributed content. Without those contributors, the book wouldn't be nearly as good as it is, and I'm deeply grateful for their help. A special thanks goes to Peter Li, who read the book from cover-to-cover and provided many fixes. I also deeply appreciate the time the reviewers (Duncan Murdoch (http://www.stats.uwo.ca/faculty/murdoch/), Karthik Ram (http://karthik.io), Vitalie Spinu (http://vitalie.spinu.info) and Ramnath Vaidyanathan (https://ramnathv.github.io)) spent reading the book and giving me thorough feedback.

Thanks go to all contributors who submitted improvements via github (in alphabetical order): @aaronwolen, @adessy, Adrien Todeschini, Andrea Cantieni, Andy Visser, @apomatix, Ben Bond-Lamberty, Ben Marwick, Brett K, Brett Klamer, @contravariant, Craig Citro, @davidkane9, David Robinson, David Smith, Dean Attali, Eduardo Ariño de la Rubia, Federico Marini, Gerhard Nachtmann, Gerrit-Jan Schutten, Hadley Wickham, Henrik Bengtsson, @heogden, Ian Gow, @jacobbien, Jennifer (Jenny) Bryan, Jim Hester, @jmarshallnz, Joanna Zhao, Jo-Anne Tan, Joe Cainey, John Blischak, @jowalski, Justin Alford, Karl Broman, Karthik Ram, Kevin Ushey, Kun Ren, @kwenzig, @kylelundstedt, @lancelote, Lech Madeyski, @lindbrook, @maiermarco, Manuel Reif, Michael Buckley, @MikeLeonard, Nick Carchedi, Oliver Keyes, Patrick Kimes, Paul Blischak, @PeterDee, Peter Meissner, Po Su, Richard M. Smith, @rmar073, R. Mark Sharp, @rmsharp, Robert Krzyzanowski, @ryanatanner, Sascha Holzhauer, @scharne, Sean Wilkinson, @SimonPBiggs, Stefan Widgren, Stephen Frank, Stephen Rushe, Tony Breyal, Tony Fischetti, @urmils, Vlad Petyuk, Winston Chang, @winterschlaefer, @wrathematics, @zhaoy.

1.5 Conventions

Throughout this book, I write `foo()` to refer to functions, `bar` to refer to variables and function parameters, and `baz/` to paths.

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

1.6 Colophon

This book was written in Rmarkdown (`http://rmarkdown.rstudio.com/`) inside RStudio (`http://www.rstudio.com/ide/`). knitr (`http://yihui.name/knitr/`) and pandoc (`http://johnmacfarlane.net/pandoc/`) converted the raw Rmarkdown to html and pdf. The website (`http://r-pkgs.had.co.nz`) was made with jekyll (`http://jekyllrb.com/`), styled with bootstrap (`http://getbootstrap.com/`), and automatically published to Amazon's S3 (`http://aws.amazon.com/s3/`) by travis-ci (`https://travis-ci.org/`). The complete source is available from github (`https://github.com/hadley/r-pkgs`).

This version of the book was built with:

```
library(roxygen2)
library(testthat)
devtools::session_info()
#> Session info ------------------------------------------------
#>  setting  value
#>  version  R version 3.2.0 (2015-04-16)
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  en_US.UTF-8
#>  tz       <NA>
#> Packages ----------------------------------------------------
#>  package   * version date
#>  bookdown  * 0.1     2015-06-26
#>  crayon      1.3.0   2015-06-05
#>  curl        0.9     2015-06-19
#>  devtools    1.8.0   2015-05-09
#>  digest      0.6.8   2014-12-31
#>  evaluate    0.7     2015-04-21
#>  formatR     1.2     2015-04-21
#>  git2r       0.10.1  2015-05-07
#>  htmltools   0.2.6   2014-09-08
#>  knitr        1.10.5  2015-05-06
#>  magrittr     1.5     2014-11-22
#>  memoise      0.2.1   2014-04-22
#>  Rcpp         0.11.6  2015-05-01
#>  rmarkdown    0.7     2015-06-26
#>  roxygen2   * 4.1.1   2015-04-15
#>  rversions    1.0.1   2015-06-06
#>  stringi      0.5-2   2015-06-22
#>  stringr      1.0.0   2015-04-30
#>  testthat   * 0.10.0  2015-05-22
#>  xml2         0.1.1   2015-06-02
#>  yaml         2.1.13  2014-06-12
#>  source
#>  Github (hadley/bookdown@1b83ee1)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  Github (rstudio/rmarkdown@82fde67)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
#>  CRAN (R 3.2.0)
```

Chapter 2

Package structure

This chapter will start you on the road to package development by showing you how to create your first package. You'll also learn about the various states a package can be in, including what happens when you install a package. Finally, you'll learn about the difference between a package and a library and why you should care.

2.1 Naming your package

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

Before you can create your first package, you need to come up with a name for it. I think this is the hardest part of creating a package! (Not least because devtools can't automate it for you.)

2.1.1 Requirements for a name

There are three formal requirements: the name can only consist of letters, numbers and periods, i.e., `.`; it must start with a letter; and it cannot end with a period. Unfortunately, this means you can't use either hyphens or underscores, i.e., `-` or `_`, in your package name. I recommend against using periods in package names because it has confusing connotations (i.e., file extension or S3 method).

2.1.2 Strategies for creating a name

If you're planning on releasing your package, I think it's worth spending a few minutes to come up with a good name. Here are some recommendations for how to go about it:

- Pick a unique name you can easily Google. This makes it easy for potential users to
  find your package (and associated resources) and for you to see who's using it. You can also check if a name is already used on CRAN by loading `http://cran.r-project.org/web/packages/{[}PACKAGE_NAME{]}`.
- Avoid using both upper and lower case letters: doing so makes the package name hard to type and even harder to remember. For example, I can never remember if it's Rgtk2 or RGTK2 or RGtk2.
- Find a word that evokes the problem and modify it so that it's unique:
  - plyr is generalisation of the apply family, and evokes pliers.
  - lubridate makes dates and times easier.
  - knitr (knit + r) is "neater" than sweave (s + weave).
  - testdat tests that data has the correct format.
- Use abbreviations:
  - Rcpp = R + C++ (plus plus)
  - lvplot = letter value plots.
- Add an extra R:
  - stringr provides string tools.
  - tourr implements grand tours (a visualisation method).
  - gistr lets you programmatically create and modify GitHub gists.

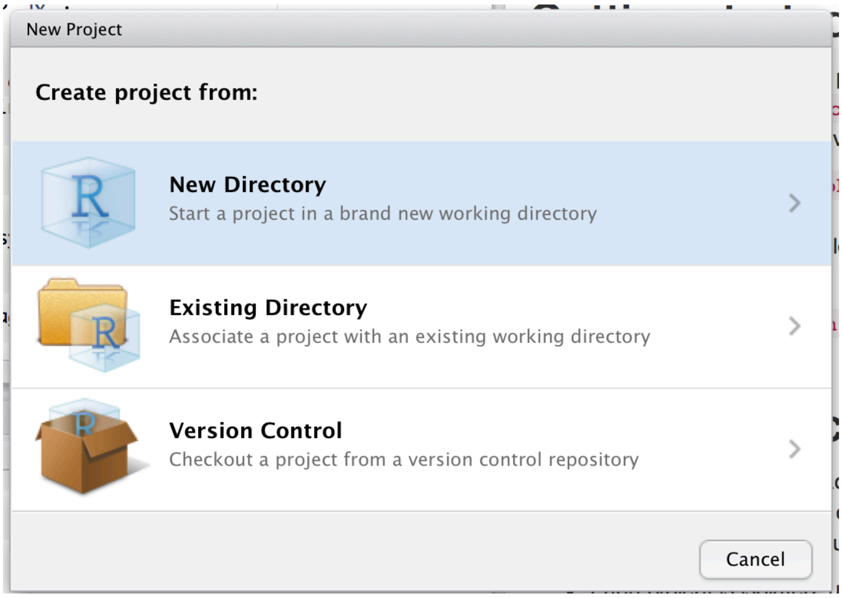
If you're creating a package that talks to a commercial service, make sure you check the branding guidelines to avoid problems down the line. For example, rDrop isn't called rDropbox because Dropbox prohibits any applications from using the full trademarked name.

2.2 Creating a package

Once you've come up with a name, there are two ways to create the package. You can use RStudio:

1. Click File | New Project.

2. Choose “New Directory”:



New Project

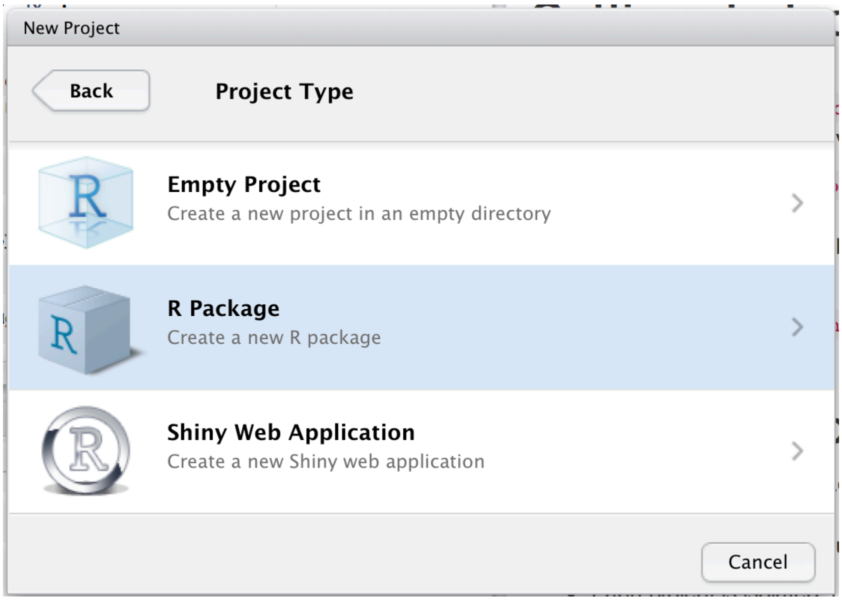
Create project from:

New Directory
Start a project in a brand new working directory

Existing Directory
Associate a project with an existing working directory

Version Control
Checkout a project from a version control repository

3. Then “R Package”:



New Project

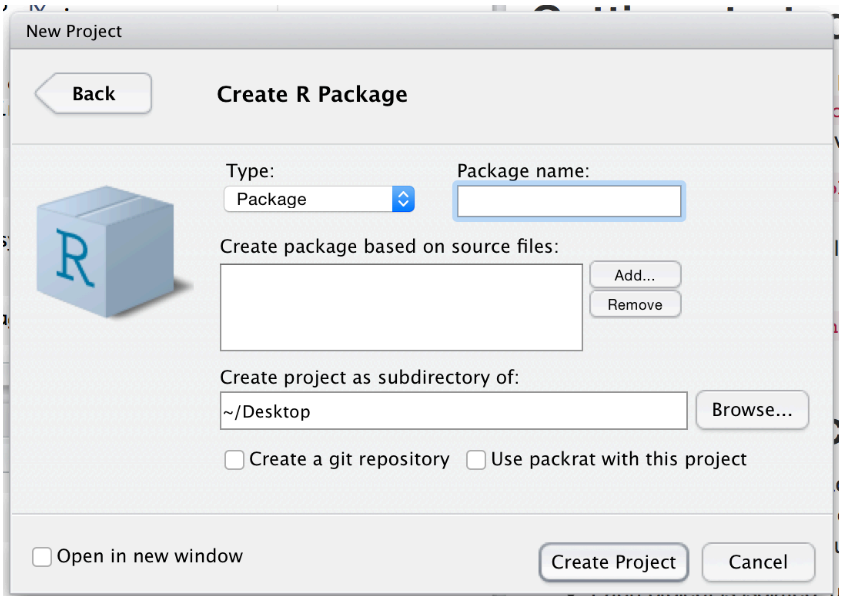
Project Type

Empty Project
Create a new project in an empty directory

R Package
Create a new R package

Shiny Web Application
Create a new Shiny web application

4. Then give your package a name and click “Create Project”:



New Project

Create R Package

Type: Package

Package name:

Create package based on source files:

Create project as subdirectory of: ~/Desktop

□ Create a git repository □ Use packrat with this project

□ Open in new window

Alternatively, you can create a new package from within R by running

```
devtools::create("path/to/package/pkgname")
```

Either route gets you to the same place: the smallest usable package, one with three components:

1. An `R/` directory, which you'll learn about in Chapter 3.
2. A basic `DESCRIPTION` file, which you'll learn about in Chapter 4.
3. A basic `NAMESPACE` file, which you'll learn about in Chapter 8.

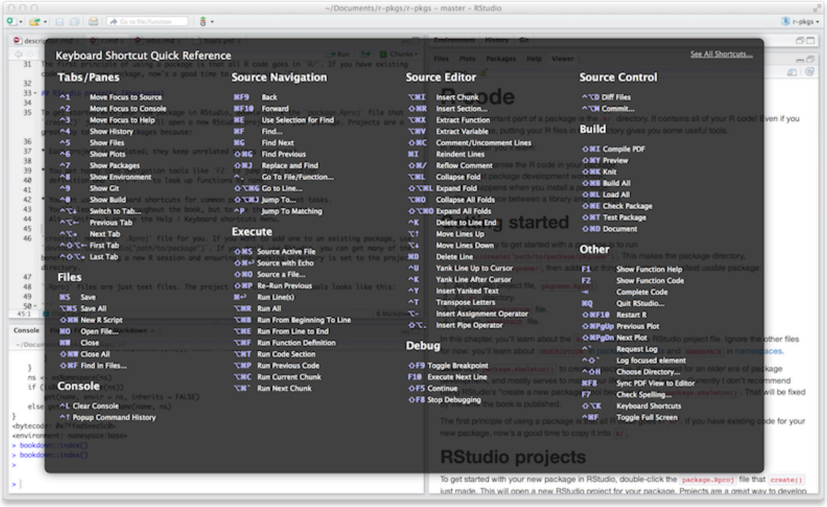
It will also include an RStudio project file, `pkgname.Rproj`, that makes your package easy to use with RStudio, as described below.

Don't use `package.skeleton()` to create a package. Following that workflow actually creates more work for you because it creates extra files that you'll have to delete or modify before you can have a working package.

2.3 RStudio projects

To get started with your new package in RStudio, double-click the `pkgname.Rproj` file that `create()` just made. This will open a new RStudio project for your package. Projects are a great way to develop packages because:

- Each project is isolated; code run in one project does not affect any other project.
- You get handy code navigation tools like `F2` to jump to a function definition and `Ctrl + .` to look up functions by name.
- You get useful keyboard shortcuts for common package development tasks. You'll learn about them throughout the book. But to see them all, press Alt + Shift + K or use the Help | Keyboard shortcuts menu.



Keyboard Shortcut Quick Reference

Tabs/Panes

^1	Move Focus to Source
^2	Move Focus to Console
^3	Move Focus to Help
^4	Show History
^5	Show Files
^6	Show Plots
^7	Show Packages
^8	Show Environment
^9	Show Git
^0	Show Build
^⌥↓	Switch to Tab...
^⌥←	Previous Tab
^⌥→	Next Tab
^⇧⌥←	First Tab
^⇧⌥→	Last Tab

Files

⌘S	Save
⌥⌘S	Save All
⇧⌘N	New R Script
⌘O	Open File...
⌘W	Close
⇧⌘W	Close All
⇧⌘F	Find in Files...

Console

^L	Clear Console
^↑	Popup Command History

Source Navigation

⌘F9	Back
⌘F10	Forward
⌘E	Use Selection for Find
⌘F	Find...
⌘G	Find Next
⇧⌘G	Find Previous
⇧⌘J	Replace and Find
^.	Go To File/Function...
⇧⌥⌘G	Go to Line...
⇧⌥⌘J	Jump To...
^P	Jump To Matching

Execute

⇧⌘S	Source Active File
⇧⌘↩	Source with Echo
⇧⌘O	Source a File...
⇧⌘P	Re-Run Previous
⌘↩	Run Line(s)
⌥⌘R	Run All
⌥⌘B	Run From Beginning To Line
⌥⌘E	Run From Line to End
⌥⌘F	Run Function Definition
⌥⌘T	Run Code Section
⌥⌘P	Run Previous Code
⌥⌘C	Run Current Chunk
⌥⌘`	Run Next Chunk

Source Editor

⌥⌘I	Insert Chunk
⇧⌘R	Insert Section...
⌥⌘X	Extract Function
⌥⌘V	Extract Variable
⇧⌘C	Comment/Uncomment Lines
⌘I	Reindent Lines
⇧⌘/	Reflow Comment
⌥⌘L	Collapse Fold
⇧⌥⌘L	Expand Fold
⌥⌘O	Collapse All Folds
⇧⌥⌘O	Expand All Folds
^K	Delete to Line End
⌥↑	Move Lines Up
⌥↓	Move Lines Down
⌘D	Delete Line
^U	Yank Line Up to Cursor
^K	Yank Line After Cursor
^Y	Insert Yanked Text
^T	Transpose Letters
⌥-	Insert Assignment Operator
⇧⌥.	Insert Pipe Operator

Debug

⇧F9	Toggle Breakpoint
F10	Execute Next Line
⇧F5	Continue
⇧F8	Stop Debugging

Source Control

^⌥D	Diff Files
^⌥M	Commit...

Build

⇧⌘I	Compile PDF
⇧⌘Y	Preview
⇧⌘K	Knit
⇧⌘B	Build All
⇧⌘L	Load All
⇧⌘E	Check Package
⇧⌘T	Test Package
⇧⌘D	Document

Other

F1	Show Function Help
F2	Show Function Code
⇥	Complete Code
⌘Q	Quit RStudio...
⇧⌘F10	Restart R
⇧⌘PgUp	Previous Plot
⇧⌘PgDn	Next Plot
^`	Request Log
^⇧`	Log focused element
^⇧H	Choose Directory...
⌘F8	Sync PDF View to Editor
F7	Check Spelling...
⇧⌥K	Keyboard Shortcuts
^⌘F	Toggle Full Screen

(If you want to learn more RStudio tips and tricks, follow @rstudiotips (`https://twitter.com/rstudiotips`) on twitter.)

Both RStudio and `devtools::create()` will make an `.Rproj` file for you. If you have an existing package that doesn't have an `.Rproj` file, you can use `devtools::use_rstudio("path/to/package")` to add it. If you don't use RStudio, you can get many of the benefits by starting a new R session and ensuring the working directory is set to the package directory.

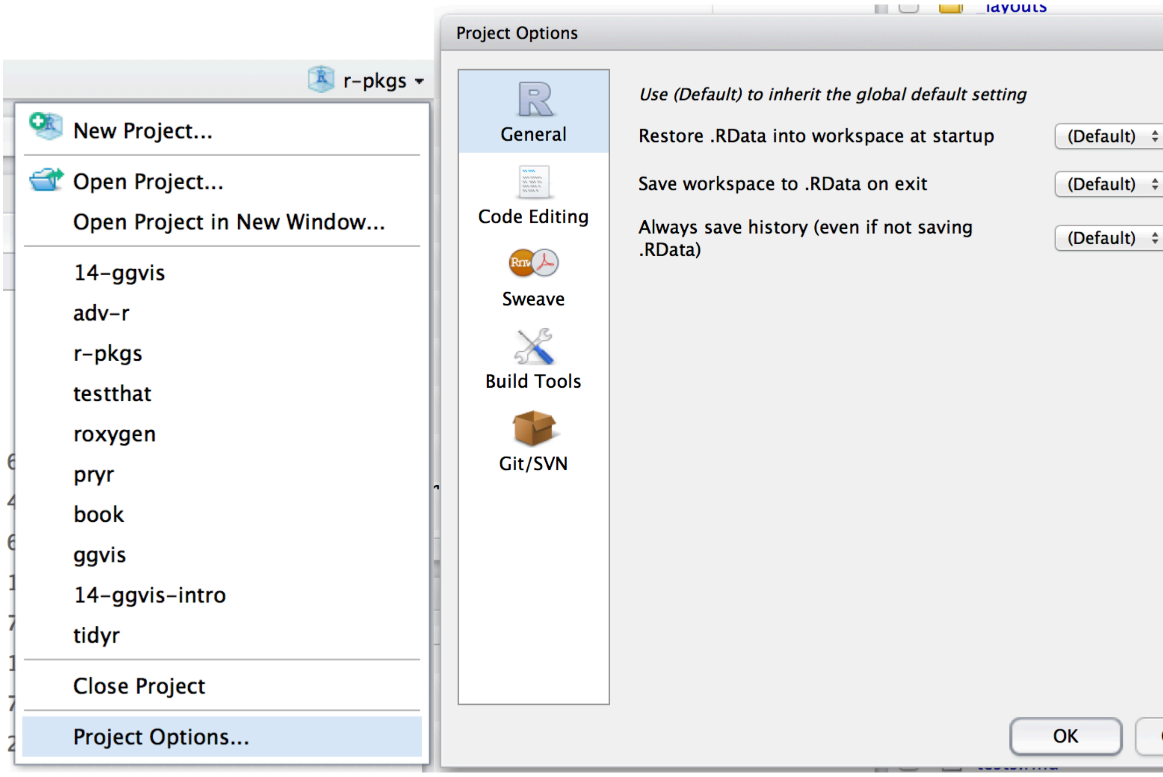
2.3.1 What is an RStudio project file?

An `.Rproj` file is just a text file. The project file created by devtools looks like this:

```
Version: 1.0

RestoreWorkspace: No
SaveWorkspace: No
AlwaysSaveHistory: Default
```

```
EnableCodeIndexing: Yes
Encoding: UTF-8

AutoAppendNewline: Yes
StripTrailingWhitespace: Yes

BuildType: Package
PackageUseDevtools: Yes
PackageInstallArgs: --no-multiarch --with-keep.source
PackageRoxygenize: rd,collate,namespace
```

You don't need to modify this file by hand. Instead, use the friendly project options dialog box, accessible from the projects menu in the top-right corner of RStudio.



Use (Default) to inherit the global default setting

Restore .RData into workspace at startup (Default)

Save workspace to .RData on exit (Default)

Always save history (even if not saving .RData) (Default)

2.4 What is a package?

To make your first package, all you need to know is what you've learnt above. To master package development, particularly when you're distributing a package to others, it really helps to understand the five states a package can be in across its lifecycle: source, bundled, binary, installed and in-memory. Understanding the differences between these states will help you form a better mental model of what `install.packages()` and `devtools::install_github()` do, and will make it easier to debug problems when they arise.

2.4.1 Source packages

So far, we've just worked with a **source** package: the development version of a package that lives on your computer. A source package is just a directory with components like `R/`, `DESCRIPTION`, and so on.

2.4.2 Bundled packages

A **bundled** package is a package that's been compressed into a single file. By convention (from Linux), package bundles in R use the extension `.tar.gz`. This means that multiple files have been reduced to a single file (`.tar`) and then compressed using gzip (`.gz`). While a bundle is not that useful on its own, it's a useful intermediary between the other states. In the rare case that you do need a bundle, call `devtools::build()` to make it.

If you decompress a bundle, you'll see it looks almost the same as your source package. The main differences between an uncompressed bundle and a source package are:

- Vignettes are built so that you get HTML and PDF output instead of Markdown or LaTeX input.
- Your source package might contain temporary files used to save time during development, like compilation artefacts in `src/`. These are never found in a bundle.
- Any files listed in `.Rbuildignore` are not included in the bundle.

`.Rbuildignore` prevents files in the source package from appearing in the bundled package. It allows you to have additional directories in your source package that will not be included in the package bundle. This is particularly useful when you generate package contents (e.g. data) from other files. Those files should be included in the source package, but only the results need to be distributed. This is particularly important for CRAN packages (where the set of allowed top-level directories is fixed). Each line gives a Perl-compatible regular expression that is matched, without regard to case, against the path to each file (i.e. `dir(full.names = TRUE)` run from the package root directory) - if the regular expression matches, the file is excluded.

If you wish to exclude a specific file or directory (the most common use case), you **MUST** anchor the regular expression. For example, to exclude a directory called notes, use `^notes$`. The regular expression `notes` will match any file name containing notes, e.g. `R/notes.R`, `man/important-notes.R`, `data/endnotes.Rdata`, etc. The safest way to exclude a specific file or directory is to use `devtools::use_build_ignore("notes")`, which does the escaping for you.

Here's a typical `.Rbuildignore` file from one of my packages:

```
^.*\.Rproj$         # Automatically added by RStudio,
^\.Rproj\.user$     # used for temporary files.
^README\.Rmd$       # An Rmarkdown file used to generate README.md
^cran-comments\.md$ # Comments for CRAN submission
^NEWS\.md$          # A news file written in Markdown
^\.travis\.yml$     # Used for continuous integration testing with travis
```

I'll mention when you need to add files to `.Rbuildignore` whenever it's important.

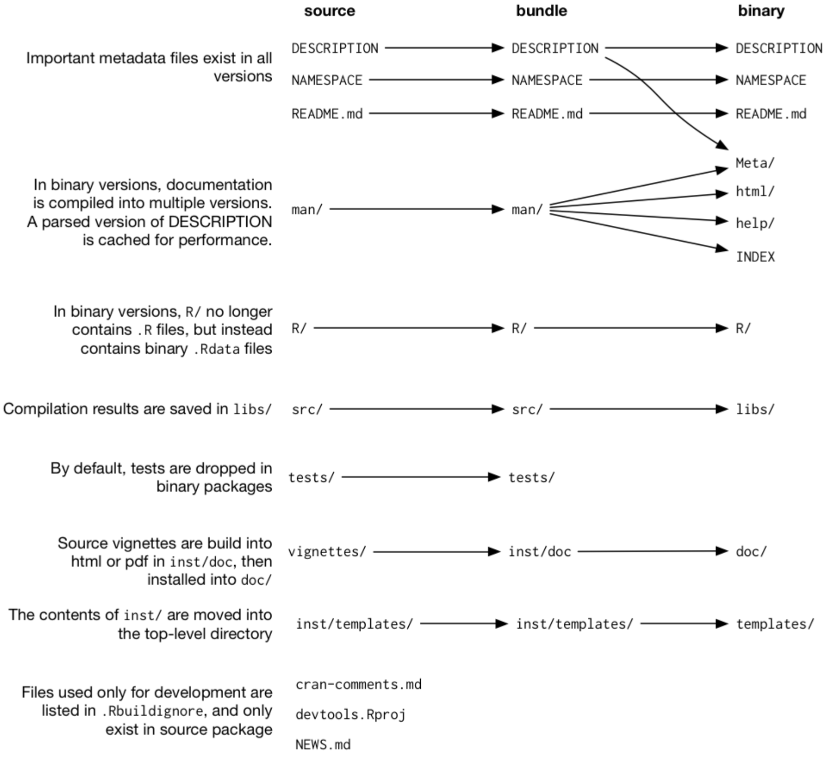
2.4.3 Binary packages

If you want to distribute your package to an R user who doesn't have package development tools, you'll 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 rather different from a source package:

- There are no `.R` files in the `R/` directory - instead there are three files that store the parsed functions in an efficient file format. This is basically the result of loading all the R code and then saving the functions with `save()`. (In the process, this adds a little extra metadata to make things as fast as possible).
- A `Meta/` directory contains a number of `Rds` files. These files contain cached metadata about the package, like what topics the help files cover and parsed versions of the `DESCRIPTION` files. (You can use `readRDS()` to see exactly what's in those files). These files make package loading faster by caching costly computations.
- An `html/` directory contains files needed for HTML help.
- If you had any code in the `src/` directory there will now be a `libs/` directory that contains the results of compiling 32 bit (`i386/`) and 64 bit (`x64/`) code.
- The contents of `inst/` are moved to the top-level directory.

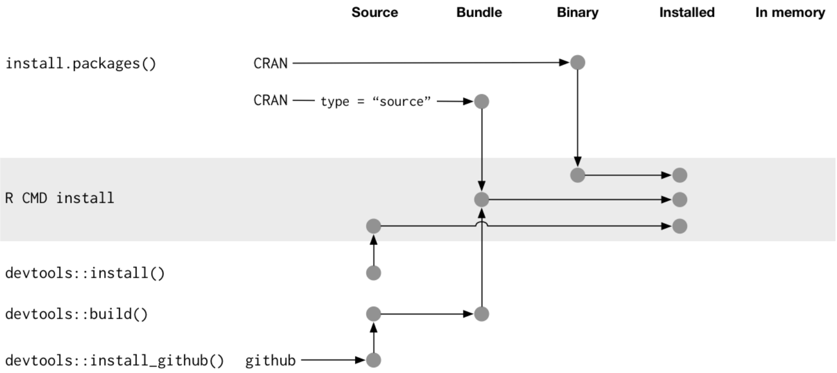
Binary packages are platform specific: you can't install a Windows binary package on a Mac or vice versa. Also, while Mac binary packages end in `.tgz`, Windows binary packages end in `.zip`. You can use `devtools::build(binary = TRUE)` to make a binary package.

The following diagram summarises the files present in the root directory for source, bundled and binary versions of devtools.


source
bundle
binary
Important metadata files exist in all versions
DESCRIPTION
NAMESPACE
README.md
DESCRIPTION
NAMESPACE
README.md
DESCRIPTION
NAMESPACE
README.md
Meta/
In binary versions, documentation is compiled into multiple versions. A parsed version of DESCRIPTION is cached for performance.
man/
man/
html/
help/
INDEX
In binary versions, R/ no longer contains .R files, but instead contains binary .Rdata files
R/
R/
R/
Compilation results are saved in libs/
src/
src/
libs/
By default, tests are dropped in binary packages
tests/
tests/
Source vignettes are build into html or pdf in inst/doc, then installed into doc/
vignettes/
inst/doc
doc/
The contents of inst/ are moved into the top-level directory
inst/templates/
inst/templates/
templates/
Files used only for development are listed in .Rbuildignore, and only exist in source package
cran-comments.md
devtools.Rproj
NEWS.md

2.4.4 Installed packages

An **installed** package is just a binary package that's been decompressed into a package library (described below). The following diagram illustrates the many ways a package can be installed. This diagram is complicated! In an ideal world, installing a package would involve stringing together a set of simple steps: source -> bundle, bundle -> binary, binary -> installed. In the real world, it's not this simple because there are often (faster) shortcuts available.



The tool that powers all package installation is the command line tool `R CMD INSTALL` - it can install a source, bundle or a binary package. Devtools functions provide wrappers that allow you to access this tool from R rather than from the command line. `devtools::install()` is effectively a wrapper for `R CMD INSTALL`. `devtools::build()` is a wrapper for `R CMD build` that turns source packages into bundles. `devtools::install_github()` downloads a source package from GitHub, runs `build()` to make vignettes, and then uses `R CMD INSTALL` to do the install. `devtools::install_url()`, `devtools::install_gitorious()`, `devtools::install_bitbucket()` work similarly for packages found elsewhere on the internet.

`install.packages()` and `devtools::install_github()` allow you to install a remote package. Both work by downloading and then installing the package. This makes installation very speedy. `install.packages()` is used to download and install binary packages built by CRAN. `install_github()` works a little differently - it downloads a source package, builds it and then installs it.

You can prevent files in the package bundle from being included in the installed package using `.Rinstignore`. This works the same way as `.Rbuildignore`, described above. It's rarely needed.

2.4.5 In memory packages

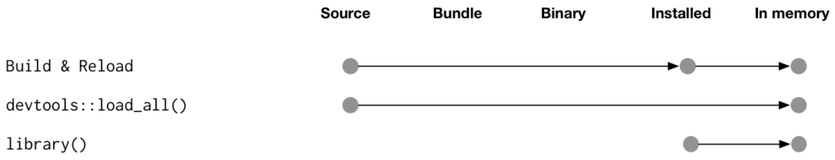
To use a package, you must load it into memory. To use it without providing the package name (e.g. `install()` instead of `devtools::install()`), you need to attach it to the search path. R loads packages automatically when you use them. `library()` (and the later discussed `require()`) load, then attach an installed package.

```
# Automatically loads devtools
devtools::install()

# Loads and _attaches_ devtools to the search path
```

```
library(devtools)
install()
```

The distinction between loading and attaching packages is not important when you're writing scripts, but it's very important when you're writing packages. You'll learn more about the difference and why it's important in Section 8.2.

`library()` is not useful when you're developing a package because you have to install the package first. In future chapters you'll learn about `devtools::load_all()` and RStudio's "Build and reload", which allows you to skip install and load a source package directly into memory:



2.5 What is a library?

A library is simply a directory containing installed packages. You can have multiple libraries on your computer. In fact, almost every one has at least two: one for packages you've installed, and one for the packages that come with every R installation (like base, stats, etc). Normally, the directories with user-installed packages vary based on the version of R that you're using. That's why it seems like you lose all of your packages when you reinstall R — they're still on your hard drive, but R can't find them.

You can use `.libPaths()` to see which libraries are currently active. Here are mine:

```
.libPaths()
#> [1] "/Users/hadley/R"
#> [2] "/Library/Frameworks/R.framework/Versions/3.1/Resources/library"
lapply(.libPaths(), dir)
#> [[1]]
#>   [1] "AnnotationDbi"   "ash"              "assertthat"
#>   ...
#> [163] "xtable"          "yaml"             "zoo"
#>
#> [[2]]
#>  [1] "base"         "boot"         "class"        "cluster"
#>  [5] "codetools"    "compiler"     "datasets"     "foreign"
#>  [9] "graphics"     "grDevices"    "grid"         "KernSmooth"
#> [13] "lattice"      "MASS"         "Matrix"       "methods"
#> [17] "mgcv"         "nlme"         "nnet"         "parallel"
#> [21] "rpart"        "spatial"      "splines"      "stats"
#> [25] "stats4"       "survival"     "tcltk"        "tools"
#> [29] "translations" "utils"
```

The first lib path is for the packages I've installed (I've installed at lot!). The second is for so-called "recommended" packages that come with every installation of R.

When you use `library(pkg)` or `require(pkg)` to load a package, R looks through each path in `.libPaths()` to see if a directory called `pkg` exists. If it doesn't exist, you'll get an error message:

```
library(blah)
#> Error in library(blah): there is no package called 'blah'
# or:
require(blah)
#> Loading required package: blah
#> Warning in library(package, lib.loc = lib.loc, character.only
#> = TRUE, logical.return = TRUE, : there is no package called
#> 'blah'
```

The main difference between `library()` and `require()` is what happens if a package isn't found. While `library()` throws an error, `require()` prints a warning message and returns FALSE. In practice, this distinction isn't important because when building a package you should **NEVER** use either inside a package. See Section 4.1 for what you should do instead.

When you start learning R, it's easy to get confused between libraries and packages because you use `library()` function to load a *package*. However, the distinction between the two is important and useful. For example, one important application is packrat, which automates the process of managing project specific libraries. With packrat, when you upgrade a package in one project, it only affects that project, not every project on your computer. This is useful because it allows you to play around with cutting-edge packages without affecting other projects' use of older, more reliable packages. This is also useful when you're both developing and using a package.

Part II

Package components

Chapter 3

R code

The first principle of using a package is that all R code goes in `R/`. In this chapter, you'll learn about the `R/` directory, my recommendations for organising your functions into files, and some general tips on good style. You'll also learn about some important differences between functions in scripts and functions in packages.

3.1 R code workflow

The first practical advantage to using a package is that it's easy to re-load your code. You can either run `devtools::load_all()`, or in RStudio press **Ctrl/Cmd + Shift + L**, which also saves all open files, saving you a keystroke.

This keyboard shortcut leads to a fluid development workflow:

1. Edit an R file.

2. Press Ctrl/Cmd + Shift + L.

3. Explore the code in the console.

4. Rinse and repeat.

Congratulations! You've learned your first package development workflow. Even if you learn nothing else from this book, you'll have gained a useful workflow for editing and reloading R code.

3.2 Organising your functions

While you're free to arrange functions into files as you wish, the two extremes are bad: don't put all functions into one file and don't put each function into its own separate file. (It's OK if some files only contain one function, particularly if the function is large or has a lot of documentation.). File names should be meaningful and end in `.R`.

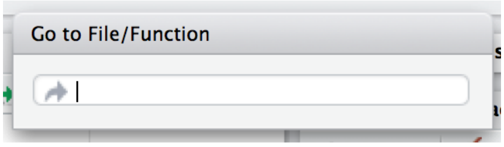
```
# Good
fit_models.R
utility_functions.R

# Bad
foo.r
stuff.r
```

Pay attention to capitalization, since you, or some of your collaborators, might be using an operating system with a case-insensitive file system (e.g., Microsoft Windows or OS X). Avoid problems by never using filenames that differ only in capitalisation.

My rule of thumb is that if I can't remember the name of the file where a function lives, I need to either separate the functions into more files or give the file a better name. (Unfortunately you can't use subdirectories inside `R/`. The next best thing is to use a common prefix, e.g., `abc-*.R`.).

The arrangement of functions within files is less important if you master two important RStudio keyboard shortcuts that let you jump to the definition of a function:

- Click a function name in code and press **F2**.
- Press **Ctrl + .** then start typing the name:



After navigating to a function using one of these tools, you can go back to where you were by clicking the back arrow at the top-left of the editor (⇦ ⇨), or by pressing Ctrl/Cmd-F9.

3.3 Code style

Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read. As with styles of punctuation, there are many possible variations. The following guide describes the style that I use (in this book and elsewhere). It is based on Google's R style guide (`https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml`), with a few tweaks.

You don't have to use my style, but I strongly recommend that you use a consistent style and you document it. If you're working on someone else's code, don't impose your own style. Instead, read their style documentation and follow it as closely as possible.

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

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

```
install.packages("formatR")
formatR::tidy_dir("R")
```

A complementary approach is to use a code **linter**. Rather than automatically fixing problems, a linter just warns you about them. The lintr package by Jim Hester checks for compliance with this style guide and lets you know where you've missed something. Compared to formatR, it picks up more potential problems (because it doesn't need to also fix them), but you will still see false positives.

```
install.packages("lintr")
lintr::lint_package()
```

3.3.1 Object names

Variable and function names should be lowercase. Use an underscore (`_`) to separate words within a name (reserve `.` for S3 methods). Camel case is a legitimate alternative, but be consistent! Generally, variable names should be nouns and function names should be verbs. Strive for names that are concise and meaningful (this is not easy!).

```
# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1
```

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

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

3.3.2 Spacing

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

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

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

There's a small exception to this rule: `:`, `::` and `:::` don't need spaces around them. (If you haven't seen `::` or `:::` before, don't worry - you'll learn all about them in Chapter 8.)

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

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

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

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

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

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

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

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

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

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

3.3.3 Curly braces

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

Always indent the code inside curly braces.

```
# Good

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

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

# Bad

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

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

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

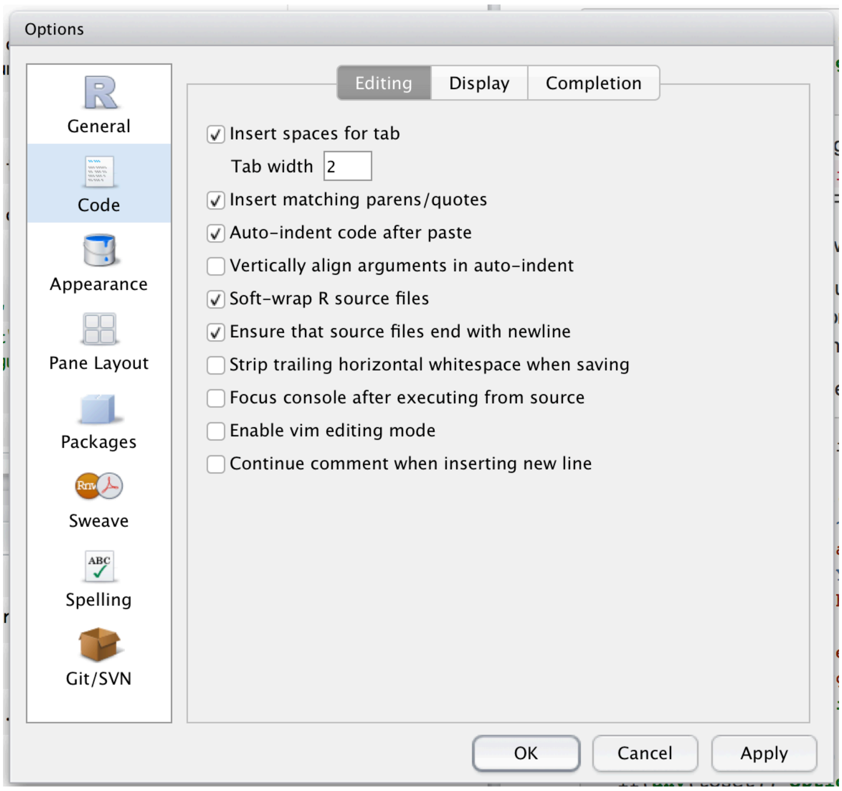
```
if (y < 0 && debug) message("Y is negative")
```

3.3.4 Line length

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

3.3.5 Indentation

When indenting your code, use two spaces. Never use tabs or mix tabs and spaces. Change these options in the code preferences pane:



☑ Insert spaces for tab

Tab width 2

☑ Insert matching parens/quotes

☑ Auto-indent code after paste

☐ Vertically align arguments in auto-indent

☑ Soft-wrap R source files

☑ Ensure that source files end with newline

☐ Strip trailing horizontal whitespace when saving

☐ Focus console after executing from source

☐ Enable vim editing mode

☐ Continue comment when inserting new line

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

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

3.3.6 Assignment

Use <-, not =, for assignment.

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

3.3.7 Commenting guidelines

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

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

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

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

3.4 Top-level code

Up until now, you've probably been writing **scripts**, R code saved in a file that you load with `source()`. There are two main differences between code in scripts and packages:

- In a script, code is run when it is loaded. In a package, code is run when it is built. This means your package code should only create objects, the vast majority of which will be functions.
- Functions in your package will be used in situations that you didn't imagine. This means your functions need to be thoughtful in the way that they interact with the outside world.

The next two sections expand on these important differences.

3.4.1 Loading code

When you load a script with `source()`, every line of code is executed and the results are immediately made available. Things are different in a package, because it is loaded in two steps. When the package is built (e.g. by CRAN) all the code in `R/` is executed and the results are saved. When you load a package, with `library()` or `require()`, the cached results are made available to you. If you loaded scripts in the same way as packages, your code would look like this:

```
# Load a script into a new environment and save it
env <- new.env(parent = emptyenv())
source("my-script.R", local = env)
save(envir = env, "my-script.Rdata")

# Later, in another R session
load("my-script.Rdata")
```

For example, take `x <- Sys.time()`. If you put this in a script, `x` would tell you when the script was `source()`d. But if you put that same code in a package, `x` would tell you when the package was *built*.

This means that you should never run code at the top-level of a package: package code should only create objects, mostly functions. For example, imagine your foo package contains this code:

```
library(ggplot2)

show_mtcars <- function() {
  qplot(mpg, wt, data = mtcars)
}
```

If someone tries to use it:

```
library(foo)
show_mtcars()
```

The code won't work because ggplot2's `qplot()` function won't be available: `library(foo)` doesn't re-execute `library(gpplot2)`. The top-level R code in a package is only executed when the package is built, not when it's loaded.

To get around this problem you might be tempted to do:

```
show_mtcars <- function() {
  library(ggplot2)
  qplot(mpg, wt, data = mtcars)
}
```

That's also problematic, as you'll see below. Instead, describe the packages your code needs in the `DESCRIPTION` file, as you'll learn in Section 4.1.

3.4.2 The R landscape

Another big difference between a script and a package is that other people are going to use your package, and they're going to use it in situations that you never imagined. This means you need to pay attention to the R landscape, which includes not just the available functions and objects, but all the global settings. You have changed the R landscape if you've loaded a package with `library()`, or changed a global option with `options()`, or modified the working directory with `setwd()`. If the behaviour of *other* functions differs before and after running your function, you've modified the landscape. Changing the landscape is bad because it makes code much harder to understand.

There are some functions that modify global settings that you should never use because there are better alternatives:

- **Don't use `library()` or `require()`.** These modify the search path, affecting what functions are available from the global environment. It's better to use the `DESCRIPTION` to specify your package's requirements, as described in the next chapter. This also makes sure those packages are installed when your package is installed.
- **Never use `source()`** to load code from a file. `source()` modifies the current environment, inserting the results of executing the code. Instead, rely on `devtools::load_all()` which automatically sources all files in `R/`. If you're using `source()` to create a dataset, instead switch to `data/` as described in Chapter 9.

Other functions need to be used with caution. If you use them, make sure to clean up after yourself with `on.exit()`:

- If you modify global `options()` or graphics `par()`, save the old values and reset when you're done:

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

- Avoid modifying the working directory. If you do have to change it, make sure to change it back when you're done:

  ```
  old <- setwd(tempdir())
  on.exit(setwd(old), add = TRUE)
  ```

- Creating plots and printing output to the console are two other ways of affecting the global R environment. Often you can't avoid these (because they're important!) but it's good practice to isolate them in functions that **only** produce output. This also makes it easier for other people to repurpose your work for new uses. For example, if you separate data preparation and plotting into two functions, others can use your data prep work (which is often the hardest part!) to create new visualisations.

The flip side of the coin is that you should avoid relying on the user's landscape, which might be different to yours. For example, functions like `read.csv()` are dangerous because the value of `stringsAsFactors` argument comes from the global option `stringsAsFactors`. If you expect it to be `TRUE` (the default), and the user has set it to be `FALSE`, your code might fail.

3.4.3 When you do need side-effects

Occasionally, packages do need side-effects. This is most common if your package talks to an external system — you might need to do some initial setup when the package loads. To do that, you can use two special functions: `.onLoad()` and `.onAttach()`. These are called when the package is loaded and attached. You'll learn about the distinction between the two in Chapter 8. For now, you should always use `.onLoad()` unless explicitly directed otherwise.

Some common uses of `.onLoad()` and `.onAttach()` are:

- To display an informative message when the package loads. This might make usage conditions clear, or display useful tips. Startup messages is one place where you should use `.onAttach()` instead of `.onLoad()`. To display startup messages, always use `packageStartupMessage()`, and not `message()`. (This allows `suppressPackageStartupMessages()` to selectively suppress package startup messages).

  ```
  .onAttach <- function(libname, pkgname) {
    packageStartupMessage("Welcome to my package")
  }
  ```

- To set custom options for your package with `options()`. To avoid conflicts with other packages, ensure that you prefix option names with the name of your package. Also be careful not to override options that the user has already set.

  I use the following code in devtools to set up useful options:

  ```
  .onLoad <- function(libname, pkgname) {
    op <- options()
    op.devtools <- list(
      devtools.path = "~/R-dev",
      devtools.install.args = "",
      devtools.name = "Your name goes here",
      devtools.desc.author = '"First Last <first.last@example.com> [aut, cre]"',
      devtools.desc.license = "What license is it under?",
      devtools.desc.suggests = NULL,
      devtools.desc = list()
    )
    toset <- !(names(op.devtools) %in% names(op))
    if(any(toset)) options(op.devtools[toset])

    invisible()
  }
  ```

  Then devtools functions can use e.g. `getOption("devtools.name")` to get the name of the package author, and know that a sensible default value has already been set.

- To connect R to another programming language. For example, if you use rJava to talk to a `.jar` file, you need to call `rJava::.jpackage()`. To make

C++ classes available as reference classes in R with Rcpp modules, you call `Rcpp::loadRcppModules()`.

- To register vignette engines with `tools::vignetteEngine()`.

As you can see in the examples, `.onLoad()` and `.onAttach()` are called with two arguments: `libname` and `pkgname`. They're rarely used (they're a holdover from the days when you needed to use `library.dynam()` to load compiled code). They give the path where the package is installed (the "library"), and the name of the package.

If you use `.onLoad()`, consider using `.onUnload()` to clean up any side effects. By convention, `.onLoad()` and friends are usually saved in a file called `zzz.R`. (Note that `.First.lib()` and `.Last.lib()` are old versions of `.onLoad()` and `.onUnload()` and should no longer be used.)

3.4.4 S4 classes, generics and methods

Another type of side-effect is defining S4 classes, methods and generics. R packages capture these side-effects so they can be replayed when the package is loaded, but they need to be called in the right order. For example, before you can define a method, you must have defined both the generic and the class. This requires that the R files be sourced in a specific order. This order is controlled by the `Collate` field in the `DESCRIPTION`. This is described in more detail in Section 5.7.2.

3.5 CRAN notes

(Each chapter will finish with some hints for submitting your package to CRAN. If you don't plan on submitting your package to CRAN, feel free to ignore them!)

If you're planning on submitting your package to CRAN, you must use only ASCII characters in your `.R` files. You can still include unicode characters in strings, but you need to use the special unicode escape `"\u1234"` format. The easiest way to do that is to use `stringi::stri_escape_unicode()`:

```
x <- "This is a bullet •"
y <- "This is a bullet \u2022"
identical(x, y)
#> [1] TRUE

cat(stringi::stri_escape_unicode(x))
#> This is a bullet \u2022
```

Chapter 4

Package metadata

The job of the DESCRIPTION file is to store important metadata about your package. When you first start writing packages, you'll mostly use these metadata to record what packages are needed to run your package. However, as time goes by and you start sharing your package with others, the metadata file becomes increasingly important because it specifies who can use it (the license) and whom to contact (you!) if there are any problems.

Every package must have a DESCRIPTION. In fact, it's the defining feature of a package (RStudio and devtools consider any directory containing DESCRIPTION to be a package). To get you started, devtools::create("mypackage") automatically adds a bare-bones description file. This will allow you to start writing the package without having to worry about the metadata until you need to. The minimal description will vary a bit depending on your settings, but should look something like this:

```
Package: mypackage
Title: What The Package Does (one line, title case required)
Version: 0.1
Authors@R: person("First", "Last", email = "first.last@example.com",
                  role = c("aut", "cre"))
Description: What the package does (one paragraph)
Depends: R (>= 3.1.0)
License: What license is it under?
LazyData: true
```

(If you're writing a lot of packages, you can set global options via devtools.desc.author, devtools.desc.license, devtools.desc.suggests, and devtools.desc. See package?devtools for more details.)

DESCRIPTION uses a simple file format called DCF, the Debian control format. You can see most of the structure in the simple example below. Each line consists of a **field** name and a value, separated by a colon. When values span multiple lines, they need to be indented:

```
Description: The description of a package is usually long,
    spanning multiple lines. The second and subsequent lines
    should be indented, usually with four spaces.
```

This chapter will show you how to use the most important DESCRIPTION fields.

4.1 Dependencies: What does your package need?

It's the job of the DESCRIPTION to list the packages that your package needs to work. R has a rich set of ways of describing potential dependencies. For example, the following lines indicate that your package needs both ggvis and dplyr to work:

```
Imports:
    dplyr,
    ggvis
```

Whereas, the lines below indicate that while your package can take advantage of ggvis and dplyr, they're not required to make it work:

```
Suggests:
    dplyr,
    ggvis,
```

Both Imports and Suggests take a comma separated list of package names. I recommend putting one package on each line, and keeping them in alphabetical order. That makes it easy to skim.

Imports and Suggests differ in their strength of dependency:

- Imports: packages listed here *must* be present for your package to work. In fact, any time your package is installed, those packages will, if not already present, be installed on your computer (devtools::load_all() also checks that the packages are installed).

  Adding a package dependency here ensures that it'll be installed. However, it does *not* mean that it will be attached along with your package (i.e., library(x)). The best practice is to explicitly refer to external functions using the syntax package::function(). This makes it very easy to identify which functions live outside of your package. This is especially useful when you read your code in the future.

If you use a lot of functions from other packages this is rather verbose. There's also a minor performance penalty associated with :: (on the order of 5µs, so it will only matter if you call the function millions of times). You'll learn about alternative ways to call functions in other packages in Section 8.6.

- Suggests: your package can use these packages, but doesn't require them. You might use suggested packages for example datasets, to run tests, build vignettes, or maybe there's only one function that needs the package.

  Packages listed in Suggests are not automatically installed along with your package. This means that you need to check if the package is available before using it (use requireNamespace(x, quietly = TRUE)). There are two basic scenarios:

  ```
  # You need the suggested package for this function
  my_fun <- function(a, b) {
    if (!requireNamespace("pkg", quietly = TRUE)) {
      stop("Pkg needed for this function to work. Please install it.",
        call. = FALSE)
    }
  }

  # There's a fallback method if the package isn't available
  my_fun <- function(a, b) {
    if (requireNamespace("pkg", quietly = TRUE)) {
      pkg::f()
    } else {
      g()
    }
  }
  ```

When developing packages locally, you never need to use Suggests. When releasing your package, using Suggests is a courtesy to your users. It frees them from downloading rarely needed packages, and lets them get started with your package as quickly as possible.

The easiest way to add Imports and Suggests to your package is to use devtools::use_package(). This automatically puts them in the right place in your DESCRIPTION, and reminds you how to use them.

```
devtools::use_package("dplyr") # Defaults to imports
#> Adding dplyr to Imports
#> Refer to functions with dplyr::fun()
devtools::use_package("dplyr", "Suggests")
```

```
#> Adding dplyr to Suggests
#> Use requireNamespace("dplyr", quietly = TRUE) to test if package is
#>  installed, then use dplyr::fun() to refer to functions.
```

4.1.1 Versioning

If you need a specific version of a package, specify it in parentheses after the package name:

```
Imports:
    ggvis (>= 0.2),
    dplyr (>= 0.3.0.1)
Suggests:
    MASS (>= 7.3.0)
```

You almost always want to specify a minimum version rather than an exact version (`MASS (== 7.3.0)`). Since R can't have multiple versions of the same package loaded at the same time, specifying an exact dependency dramatically increases the chance of conflicting versions.

Versioning is most important when you release your package. Usually people don't have exactly the same versions of packages installed that you do. If someone has an older package that doesn't have a function your package needs, they'll get an unhelpful error message. However, if you supply the version number, they'll get a error message that tells them exactly what the problem is: an out of date package.

Generally, it's always better to specify the version and to be conservative about which version to require. Unless you know otherwise, always require a version greater than or equal to the version you're currently using.

4.1.2 Other dependencies

There are three other fields that allow you to express more specialised dependencies:

- `Depends`: Prior to the rollout of namespaces in R 2.14.0, `Depends` was the only way to "depend" on another package. Now, despite the name, you should almost always use `Imports`, not `Depends`. You'll learn why, and when you should still use `Depends`, in Chapter 8.

  You can also use `Depends` to require a specific version of R, e.g. `Depends: R (>= 3.0.1)`. As with packages, it's a good idea to play it safe and require a version greater than or equal to the version you're currently using. `devtools::create()` will do this for you.

In R 3.1.1 and earlier you'll also need to use `Depends: methods` if you use S4. This bug is fixed in R 3.2.0, so methods can go back to `Imports` where they belong.

- `LinkingTo`: packages listed here rely on C or C++ code in another package. You'll learn more about `LinkingTo` in Chapter 10.

- `Enhances`: packages listed here are "enhanced" by your package. Typically, this means you provide methods for classes defined in another package (a sort of reverse `Suggests`). But it's hard to define what that means, so I don't recommend using `Enhances`.

You can also list things that your package needs outside of R in the `SystemRequirements` field. But this is just a plain text field and is not automatically checked. Think of it as a quick reference; you'll also need to include detailed system requirements (and how to install them) in your README.

4.2 Title and description: What does your package do?

The title and description fields describe what the package does. They differ only in length:

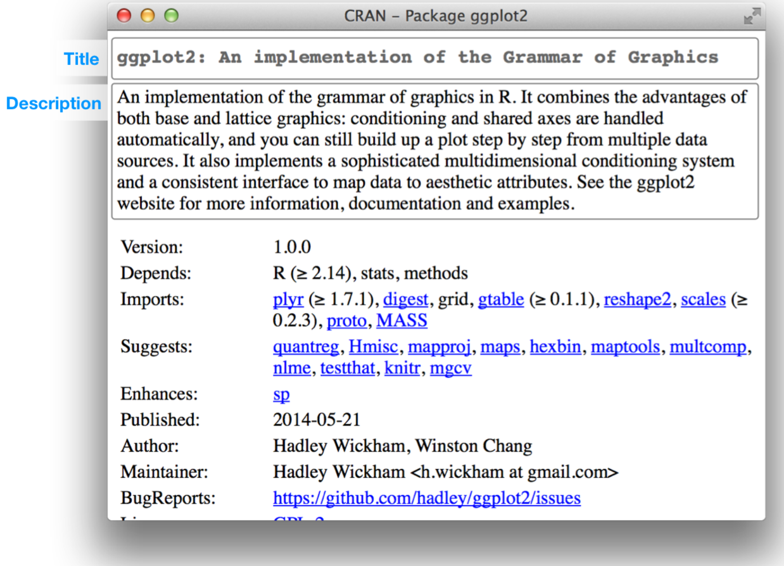
- `Title` is a one line description of the package, and is often shown in package listing. It should be plain text (no markup), capitalised like a title, and NOT end in a period. Keep it short: listings will often truncate the title to 65 characters.

- `Description` is more detailed than the title. You can use multiple sentences but you are limited to one paragraph. If your description spans multiple lines (and it should!), each line must be no more than 80 characters wide. Indent subsequent lines with 4 spaces.

The `Title` and `Description` for ggplot2 are:

```
Title: An implementation of the Grammar of Graphics
Description: An implementation of the grammar of graphics in R. It combines
    the advantages of both base and lattice graphics: conditioning and shared
    axes are handled automatically, and you can still build up a plot step
    by step from multiple data sources. It also implements a sophisticated
    multidimensional conditioning system and a consistent interface to map
    data to aesthetic attributes. See the ggplot2 website for more information,
    documentation and examples.
```

A good title and description are important, especially if you plan to release your package to CRAN because they appear on the CRAN download page as follows:



ggplot2: An implementation of the Grammar of Graphics

An implementation of the grammar of graphics in R. It combines the advantages of both base and lattice graphics: conditioning and shared axes are handled automatically, and you can still build up a plot step by step from multiple data sources. It also implements a sophisticated multidimensional conditioning system and a consistent interface to map data to aesthetic attributes. See the ggplot2 website for more information, documentation and examples.

Version:	1.0.0
Depends:	R (≥ 2.14), stats, methods
Imports:	plyr (≥ 1.7.1), digest, grid, gtable (≥ 0.1.1), reshape2, scales (≥ 0.2.3), proto, MASS
Suggests:	quantreg, Hmisc, mapproj, maps, hexbin, maptools, multcomp, nlme, testthat, knitr, mgcv
Enhances:	sp
Published:	2014-05-21
Author:	Hadley Wickham, Winston Chang
Maintainer:	Hadley Wickham <h.wickham at gmail.com>
BugReports:	https://github.com/hadley/ggplot2/issues

Because Description only gives you a small amount of space to describe what your package does, I also recommend including a README.md file that goes into much more depth and shows a few examples. You'll learn about that in Section 15.5.1.

4.3 Author: who are you?

To identify the package's author, and whom to contact if something goes wrong, use the Authors@R field. This field is unusual because it contains executable R code rather than plain text. Here's an example:

```
Authors@R: person("Hadley", "Wickham", email = "hadley@rstudio.com",
  role = c("aut", "cre"))
```

```
person("Hadley", "Wickham", email = "hadley@rstudio.com",
  role = c("aut", "cre"))
#> [1] "Hadley Wickham <hadley@rstudio.com> [aut, cre]"
```

This command says that both the author (aut) and the maintainer (cre) is Hadley Wickham, and that his email address is hadley@rstudio.com. The person() function has four main arguments:

- The name, specified by the first two arguments, `given` and `family` (these are normally supplied by position, not name). In English cultures, `given` (first name) comes before `family` (last name). In many cultures, this convention does not hold.

- The `email` address.

- A three letter code specifying the `role`. There are four important roles:

  - `cre`: the creator or maintainer, the person you should bother if you have problems.
  - `aut`: authors, those who have made significant contributions to the package.
  - `ctb`: contributors, those who have made smaller contributions, like patches.
  - `cph`: copyright holder. This is used if the copyright is held by someone other than the author, typically a company (i.e. the author's employer).

  (The full list of roles (`http://www.loc.gov/marc/relators/relaterm.html`) is extremely comprehensive. Should your package have a woodcutter ("wdc"), lyricist ("lyr") or costume designer ("cst"), rest comfortably that you can correctly describe their role in creating your package.)

If you need to add further clarification, you can also use the `comment` argument and supply the desired information in plain text.

You can list multiple authors with `c()`:

```
Authors@R: c(
    person("Hadley", "Wickham", email = "hadley@rstudio.com", role = "cre"),
    person("Winston", "Chang", email = "winston@rstudio.com", role = "aut"))
```

Alternatively, you can do this concisely by using `as.person()`:

```
Authors@R: as.person(c(
    "Hadley Wickham <hadley@rstudio.com> [aut, cre]",
    "Winston Chang <winston@rstudio.com> [aut]"
  ))
```

(This only works well for names with only one first and last name.)

Every package must have at least one author (aut) and one maintainer (cre) (they might be the same person). The creator must have an email address. These fields are used to generate the basic citation for the package (e.g. `citation("pkgname")`). Only people listed as authors will be included in the auto-generated citation. There are a few extra details if you're including code that other people have written. Since this typically occurs when you're wrapping a C library, it's discussed in Chapter 10.

As well as your email address, it's also a good idea to list other resources availble for help. You can list URLs in `URL`. Multiple URLs are separated with a comma. `BugReports` is the URL where bug reports should be submitted. For example, knitr has:

```
URL: http://yihui.name/knitr/
BugReports: https://github.com/yihui/knitr/issues
```

You can also use separate `Maintainer` and `Author` fields. I prefer not to use these fields because `Authors@R` offers richer metadata.

4.3.1 On CRAN

The most important thing to note is that your email address (i.e., the address of `cre`) is the address that CRAN will use to contact you about your package. So make sure you use an email address that's likely to be around for a while. Also, because this address will be used for automated mailings, CRAN policies require that this be for a single person (not a mailing list) and that it does not require any confirmation or use any filtering.

4.4 License: Who can use your package?

The `License` field can be either a standard abbreviation for an open source license, like `GPL-2` or `BSD`, or a pointer to a file containing more information, `file LICENSE`. The license is really only important if you're planning on releasing your package. If you don't, you can ignore this section. If you want to make it clear that your package is not open source, use `License: file LICENSE` and then create a file called `LICENSE`, containing for example:

```
Proprietary

Do not distribute outside of Widgets Incorporated.
```

Open source software licensing is a rich and complex field. Fortunately, in my opinion, there are only three licenses that you should consider for your R package:

- MIT (`https://tldrlegal.com/license/mit-license`) (v. similar: to BSD 2 and 3 clause licenses). This is a simple and permissive license. It lets people use and freely distribute your code subject to only one restriction: the license must always be distributed with the code.

  The MIT license is a "template", so if you use it, you need `License: MIT + file LICENSE`, and a `LICENSE` file that looks like this:

  ```
  YEAR: <Year or years when changes have been made>
  COPYRIGHT HOLDER: <Name of the copyright holder>
  ```

- GPL-2 (`https://tldrlegal.com/license/gnu-general-public-license-v2`) or GPL-3 (`https://tldrlegal.com/license/gnu-general-public-license-v3-(gpl-3)`). These are "copy-left" licenses. This means that anyone who distributes your code in a bundle must license the whole bundle in a GPL-compatible way. Additionally, anyone who distributes modified versions of your code (derivative works) must also make the source code available. GPL-3 is a little stricter than GPL-2, closing some older loopholes.

- CC0 (`https://tldrlegal.com/license/creative-commons-cc0-1.0-universal`). It relinquishes all your rights on the code and data so that it can be freely used by anyone for any purpose. This is sometimes called putting it in the public domain, a term which is neither well-defined nor meaningful in all countries.

  This license is most appropriate for data packages. Data, at least in the US, is not copyrightable, so you're not really giving up much. This license just makes this point clear.

If you'd like to learn more about other common licenses, Github's choosealicense.com (`http://choosealicense.com/licenses/`) is a good place to start. Another good resource is `https://tldrlegal.com/`, which explains the most important parts of each license. If you use a license other than the three I suggest, make sure you consult the "Writing R Extensions" section on licensing (`http://cran.r-project.org/doc/manuals/R-exts.html#Licensing`).

If your package includes code that you didn't write, you need to make sure you're in compliance with its license. Since this occurs most commonly when you're including C source code, it's discussed in more detail in Chapter 10.

4.4.1 On CRAN

If you want to release your package to CRAN, you must pick a standard license. Otherwise it's difficult for CRAN to determine whether or not it's legal to distribute your package! You can find a complete list of licenses that CRAN considers valid at `https://svn.r-project.org/R/trunk/share/licenses/license.db`.

4.5 Version

Formally, an R package version is a sequence of at least two integers separated by either `.` or `-`. For example, `1.0` and `0.9.1-10` are valid versions, but `1` or `1.0-devel` are not. You can parse a version number with `numeric_version`.

```
numeric_version("1.9") == numeric_version("1.9.0")
#> [1] TRUE
numeric_version("1.9.0") < numeric_version("1.10.0")
#> [1] TRUE
```

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 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.

I don't recommend taking full advantage of R's flexiblity. Instead always use `.` to separate version numbers.

- A released version number consists of three numbers, `<major>.<minor>.<patch>`. For version number 1.9.2, 1 is the major number, 9 is the minor number, and 2 is the patch number. Never use versions like `1.0`, instead always spell out the three components, `1.0.0`.

- An in-development package has a fourth component: the development version. This should start at 9000. For example, the first version of the package should be `0.0.0.9000`. There are two reasons for this recommendation: first, it makes it easy to see if a package is released or in-development, and the use of the fourth place means that you're not limited to what the next version will be. `0.0.1`, `0.1.0` and `1.0.0` are all greater than `0.0.0.9000`.

  Increment the development version, e.g. from `9000` to `9001` if you've added an important feature that another development package needs to depend on.

  If you're using svn, instead of using the arbitrary `9000`, you can embed the sequential revision identifier.

This advice here is inspired in part by Semantic Versioning (`http://semver.org`) and by the X.Org (`http://www.x.org/releases/X11R7.7/doc/xorg-docs/`

`Versions.html`) versioning schemes. Read them if you'd like to understand more about the standards of versioning used by many open source projects.

We'll come back to version numbers in the context of releasing your package, Section 15.1. For now, just remember that the first version of your package should be `0.0.0.9000`.

4.6 Other components

A number of other fields are described elsewhere in the book:

- `Collate` controls the order in which R files are sourced. This only matters if your code has side-effects; most commonly because you're using S4. This is described in more depth in Section 5.7.2.
- `LazyData` makes it easier to access data in your package. Because it's so important, it's included in the minimal description created by devtools. It's described in more detail in Chapter 9.

There are actually many other rarely, if ever, used fields. A complete list can be found in the "The DESCRIPTION file" section of the R extensions manual (`http://cran.r-project.org/doc/manuals/R-exts.html#Licensing`). You can also create your own fields to add additional metadata. The only restrictions are that you shouldn't use existing names and that, if you plan to submit to CRAN, the names you use should be valid English words (so a spell-checking NOTE won't be generated).

Chapter 5

Object documentation

Documentation is one of the most important aspects of a good package. Without it, users won't know how to use your package. Documentation is also useful for future-you (so you remember what your functions were supposed to do), and for developers extending your package.

There are multiple forms of documentation. In this chapter, you'll learn about object documentation, as accessed by `?` or `help()`. Object documentation is a type of reference documentation. It works like a dictionary: while a dictionary is helpful if you want to know what a word means, it won't help you find the right word for a new situation. Similarly, object documentation is helpful if you already know the name of the object, but it doesn't help you find the object you need to solve a given problem. That's one of the jobs of vignettes, which you'll learn about in the next chapter.

R provides a standard way of documenting the objects in a package: you write `.Rd` files in the `man/` directory. These files use a custom syntax, loosely based on LaTeX, and are rendered to HTML, plain text and pdf for viewing. Instead of writing these files by hand, we're going to use roxygen2 which turns specially formatted comments into `.Rd` files. The goal of roxygen2 is to make documenting your code as easy as possible. It has a number of advantages over writing `.Rd` files by hand:

- Code and documentation are intermingled so that when you modify your code, you're reminded to also update your documentation.
- Roxygen2 dynamically inspects the objects that it documents, so you can skip some boilerplate that you'd otherwise need to write by hand.
- It abstracts over the differences in documenting different types of objects, so you need to learn fewer details.

As well as generating .Rd files, roxygen2 can also manage your NAMESPACE and the Collate field in DESCRIPTION. This chapter discusses .Rd files and the collate field. Chapter 8 describes how you can use roxygen2 to manage your NAMESPACE, and why you should care.

5.1 The documentation workflow

In this section, we'll first go over a rough outline of the complete documentation workflow. Then, we'll dive into each step individually. There are four basic steps:

1. Add roxygen comments to your .R files.
2. Run devtools::document() (or press Ctrl/Cmd + Shift + D in RStudio) to convert roxygen comments to .Rd files. (devtools::document() calls roxygen2::roxygenise() to do the hard work.)
3. Preview documentation with ?.
4. Rinse and repeat until the documentation looks the way you want.

The process starts when you add roxygen comments to your source file: roxygen comments start with #' to distinguish them from regular comments. Here's documentation for a simple function:

```r
#' Add together two numbers.
#'
#' @param x A number.
#' @param y A number.
#' @return The sum of \code{x} and \code{y}.
#' @examples
#' add(1, 1)
#' add(10, 1)
add <- function(x, y) {
  x + y
}
```

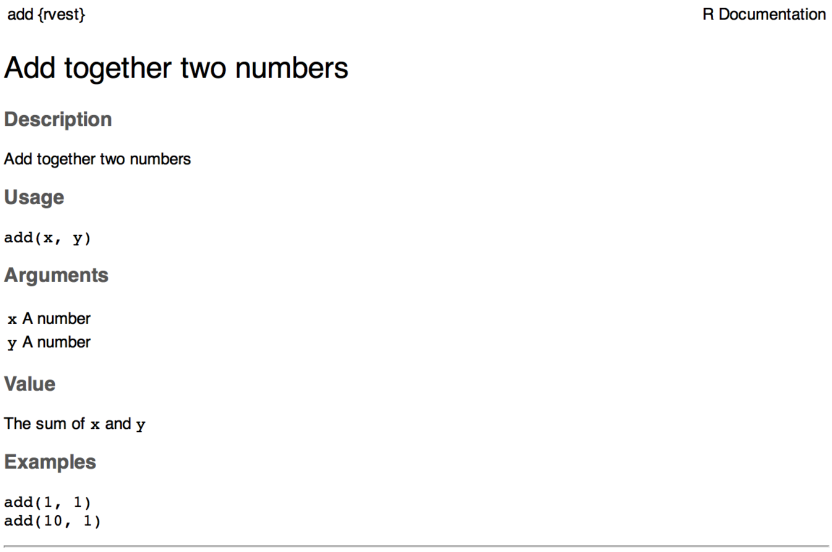
Pressing Ctrl/Cmd + Shift + D (or running devtools::document()) will generate a man/add.Rd that looks like:

```
% Generated by roxygen2 (4.0.0): do not edit by hand
\name{add}
\alias{add}
\title{Add together two numbers}
```

```
\usage{
add(x, y)
}
\arguments{
  \item{x}{A number}

  \item{y}{A number}
}
\value{
The sum of \code{x} and \code{y}
}
\description{
Add together two numbers
}
\examples{
add(1, 1)
add(10, 1)
}
```

If you're familiar with LaTeX, this should look familiar since the `.Rd` format is loosely based on it. You can read more about the Rd format in the R extensions (`http://cran.r-project.org/doc/manuals/R-exts.html#Rd-format`) manual. Note the comment at the top of the file: it was generated by code and shouldn't be modified. Indeed, if you use roxygen2, you'll rarely need to look at these files.

When you use `?add`, `help("add")`, or `example("add")`, R looks for an `.Rd` file containing `\alias{"add"}`. It then parses the file, converts it into HTML and displays it. Here's what the result looks like in RStudio:



add {rvest} R Documentation

Add together two numbers

Description

Add together two numbers

Usage

```
add(x, y)
```

Arguments

x A number
y A number

Value

The sum of x and y

Examples

```
add(1, 1)
add(10, 1)
```

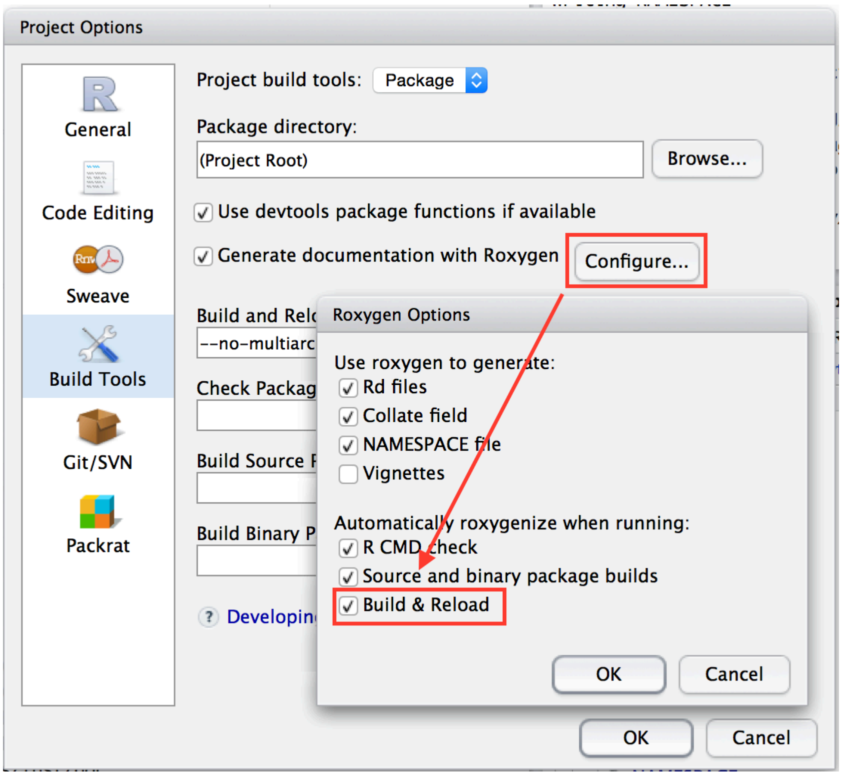
(Note you can preview development documentation because devtools overrides the usual help functions to teach them how to work with source packages. If the documentation doesn't appear, make sure that you're using devtools and that you've loaded the package with `devtools::load_all()`.)

5.2 Alternative documentation workflow

The first documentation workflow is very fast, but it has one limitation: the preview documentation pages will not show any links between pages. If you'd like to also see links, use this workflow:

1. Add roxygen comments to your `.R` files.
2. Click Build & Reload in the build pane or press Ctrl/Cmd + Shift + B. This completely rebuilds the package, including updating all the documentation, installs it in your regular library, then restarts R and reloads your package. This is slow but thorough.
3. Preview documentation with `?`.
4. Rinse and repeat until the documentation looks the way you want.

If this workflow doesn't seem to be working, check your project options in RStudio. Old versions of devtools and RStudio did not automatically update the documentation when the package was rebuilt:



Project build tools: Package

Package directory: (Project Root)

☑ Use devtools package functions if available

☑ Generate documentation with Roxygen

--no-multiarc

Use roxygen to generate:

☑ Rd files

☑ Collate field

☑ NAMESPACE file

☐ Vignettes

Automatically roxygenize when running:

☑ R CMD check

☑ Source and binary package builds

☑ Build & Reload

5.3 Roxygen comments

Roxygen comments start with `#'` and come before a function. All the roxygen lines preceding a function are called a **block**. Each line should be wrapped in the same way as your code, normally at 80 characters.

Blocks are broken up into **tags**, which look like `@tagName details`. The content of a tag extends from the end of the tag name to the start of the next tag (or the end of the block). Because `@` has a special meaning in roxygen, you need to write `@@` if you want to add a literal `@` to the documentation (this is mostly important for email addresses and for accessing slots of S4 objects).

Each block includes some text before the first tag.[1] This is called the **introduction**, and is parsed specially:

- The first *sentence* becomes the title of the documentation. That's what you see when you look at `help(package = mypackage)` and is shown at the

[1] Note that it is possible to define the "introduction" sections - the title and description - explicitly, rather than implicitly, using `@title` and `@description` tags. I follow the convention of avoiding this, as do many other package authors, so I don't recommend it - but you should be aware that these tags do exist and may appear in other package-writers' source code.

top of each help file. It should fit on one line, be written in sentence case, and end in a full stop.

- The second *paragraph* is the description: this comes first in the documentation and should briefly describe what the function does.
- The third and subsequent *paragraphs* go into the details: this is a (often long) section that is shown after the argument description and should go into detail about how the function works.

All objects must have a title and description. Details are optional.

Here's an example showing what the introduction for `sum()` might look like if it had been written with roxygen:

```r
#' 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{\link{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) {}
```

`\code{}` and `\link{}` are formatting commands that you'll learn about in Section 5.10. I've been careful to wrap the roxygen block so that it's less than 80 characters wide. You can do that automatically in Rstudio with Ctrl/Cmd + Shift + / (or from the menu, code | re-flow comment).

You can add arbitrary sections to the documentation with the `@section` tag. This is a useful way of breaking a long details section into multiple chunks with useful headings. Section titles should be in sentence case, must be followed by a colon, and they can only be one line long.

```r
#' @section Warning:
#' Do not operate heavy machinery within 8 hours of using this function.
```

There are two tags that make it easier for people to navigate between help files:

- `@seealso` allows you to point to other useful resources, either on the web, `\url{http://www.r-project.org}`, in your package `\code{\link{functioname}}`, or another package `\code{\link[packagename]{functioname}}`.
- If you have a family of related functions where every function should link to every other function in the family, use `@family`. The value of `@family` should be plural.

For sum, these components might look like:

```
#' @family aggregate functions
#' @seealso \code{\link{prod}} for products, \code{\link{cumsum}} for cumulative
#'   sums, and \code{\link{colSums}}/\code{\link{rowSums}} marginal sums over
#'   high-dimensional arrays.
```

Two other tags make it easier for the user to find documentation:

- `@aliases alias1 alias2 ...` adds additional aliases to the topic. An alias is another name for the topic that can be used with `?`.

- `@keywords keyword1 keyword2 ...` adds standardised keywords. Keywords are optional, but if present, must be taken from a predefined list found in `file.path(R.home("doc"), "KEYWORDS")`.

  Generally, keywords are not that useful except for `@keywords internal`. Using the internal keyword removes the function from the package index and disables some of their automated tests. It's common to use `@keywords internal` for functions that are of interest to other developers extending your package, but not most users.

Other tags are situational: they vary based on the type of object that you're documenting. The following sections describe the most commonly used tags for functions, packages and the various methods, generics and objects used by R's three OO systems.

5.4 Documenting functions

Functions are the most commonly documented object. As well as the introduction block, most functions have three tags: `@param`, `@examples` and `@return`.

- `@param name description` describes the function's inputs or parameters. The description should provide a succinct summary of the type of the parameter (e.g., string, numeric vector) and, if not obvious from the name, what the parameter does.

  The description should start with a capital letter and end with a full stop. It can span multiple lines (or even paragraphs) if necessary. All parameters must be documented.

  You can document multiple arguments in one place by separating the names with commas (no spaces). For example, to document both `x` and `y`, you can write `@param x,y Numeric vectors.`.

- `@examples` provides executable R code showing how to use the function in practice. This is a very important part of the documentation because many people look at the examples first. Example code must work without errors as it is run automatically as part of `R CMD check`.

  For the purpose of illustration, it's often useful to include code that causes an error. `\dontrun{}` allows you to include code in the example that is not run. (You used to be able to use `\donttest{}` for a similar purpose, but it's no longer recommended because it actually **is** tested.)

  Instead of including examples directly in the documentation, you can put them in separate files and use `@example path/relative/to/packge/root` to insert them into the documentation.

- `@return description` describes the output from the function. This is not always necessary, but is a good idea if your function returns different types of output depending on the input, or if you're returning an S3, S4 or RC object.

We could use these new tags to improve our documentation of `sum()` as follows:

```
#' 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{\link{Summary}} group generic. For this to work properly,
#' the arguments \code{...} should be unnamed, and dispatch is on the
#' first argument.
#'
#' @param ... Numeric, complex, or logical vectors.
#' @param na.rm A logical scalar. Should missing values (including NaN)
#'   be removed?
#' @return If all inputs are integer and logical, then the output
#'   will be an integer. If integer overflow
#'   \url{http://en.wikipedia.org/wiki/Integer_overflow} occurs, the output
#'   will be NA with a warning. Otherwise it will be a length-one numeric or
#'   complex vector.
#'
#'   Zero-length vectors have sum 0 by definition. See
#'   \url{http://en.wikipedia.org/wiki/Empty_sum} for more details.
#' @examples
#' sum(1:10)
#' sum(1:5, 6:10)
#' sum(F, F, F, T, T)
#'
#' sum(.Machine$integer.max, 1L)
#' sum(.Machine$integer.max, 1)
#'
#' \dontrun{
#' sum("a")
#' }
sum <- function(..., na.rm = TRUE) {}
```

Indent the second and subsequent lines of a tag so that when scanning the documentation it's easy to see where one tag ends and the next begins. Tags that always span multiple lines (like `@example`) should start on a new line and don't need to be indented.

5.5 Documenting datasets

See Section 9.1.1.

5.6 Documenting packages

You can use roxygen to provide a help page for your package as a whole. This is accessed with `package?foo`, and can be used to describe the most important components of your package. It's a useful supplement to vignettes, as described in the next chapter.

There's no object that corresponds to a package, so you need to document `NULL`, and then manually label it with `@docType package` and `@name <package-name>`. This is also an excellent place to use the `@section` tag to divide up page into useful categories.

```
#' foo: A package for computating the notorious bar statistic.
#'
#' The foo package provides three categories of important functions:
#' foo, bar and baz.
#'
#' @section Foo functions:
#' The foo functions ...
#'
#' @docType package
#' @name foo
NULL
#> NULL
```

I usually put this documentation in a file called `<package-name>.R`. It's also a good place to put the package level import statements that you'll learn about in Section 8.6.

5.7 Documenting classes, generics and methods

It's relatively straightforward to document classes, generics and methods. The details vary based on the object system you're using. The following sections give the details for the S3, S4 and RC object systems.

5.7.1 S3

S3 **generics** are regular functions, so document them as such. S3 **classes** have no formal definition, so document the constructor function. It is your choice whether or not to document S3 **methods**. You don't need to document methods for simple generics like `print()`. But if your method is more complicated or includes additional arguments, you should document it so people know how it works. In base R, you can see examples of documentation for more complex methods like `predict.lm()`, `predict.glm()`, and `anova.glm()`.

Older versions of roxygen required explicit `@method generic class` tags for all S3 methods. From version 3.0.0 onward, this is no longer needed as roxygen2 will figure it out automatically. If you are upgrading, make sure to remove these old tags. Automatic method detection will only fail if the generic and class are ambiguous. For example is `all.equal.data.frame()` the `equal.data.frame` method for `all`, or the `data.frame` method for `all.equal`?. If this happens, you can disambiguate with e.g. `@method all.equal data.frame`.

5.7.2 S4

Document **S4 classes** by adding a roxygen block before `setClass()`. Use `@slot` to document the slots of the class in the same way you use `@param` to describe the parameters of a function. Here's a simple example:

```r
#' An S4 class to represent a bank account.
#'
#' @slot balance A length-one numeric vector
Account <- setClass("Account",
  slots = list(balance = "numeric")
)
```

S4 **generics** are also functions, so document them as such. S4 **methods** are a little more complicated, however. Unlike S3, all S4 methods must be documented. You document them like a regular function, but you probably don't want each method to have its own documentation page. Instead, put the method documentation in one of three places:

- In the class. Most appropriate if the corresponding generic uses single dispatch and you created the class.
- In the generic. Most appropriate if the generic uses multiple dispatch and you have written both the generic and the method.
- In its own file. Most appropriate if the method is complex, or if you've written the method but not the class or generic.

Use either `@rdname` or `@describeIn` to control where method documentation goes. See Section 5.9.2 for details.

Another consideration is that S4 code often needs to run in a certain order. For example, to define the method `setMethod("foo", c("bar", "baz"), ...)` you must already have created the `foo` generic and the two classes. By default, R code is loaded in alphabetical order, but that won't always work for your situation. Rather than relying on alphabetic ordering, roxygen2 provides an explicit way of saying that one file must be loaded before another: `@include`. The `@include` tag gives a space separated list of file names that should be loaded before the current file:

```
#' @include class-a.R
setClass("B", contains = "A")
```

Often, it's easiest to put this at the top of the file. To make it clear that this tag applies to the whole file, and not a specific object, document `NULL`.

```
#' @include foo.R bar.R baz.R
NULL

setMethod("foo", c("bar", "baz"), ...)
```

Roxygen uses the `@include` tags to compute a topological sort which ensures that dependencies are loaded before they're needed. It then sets the `Collate` field in `DESCRIPTION`, which overrides the default alphabetic ordering. A simpler alternative to `@include` is to define all classes and methods in `aaa-classes.R` and `aaa-generics.R`, and rely on these coming first since they're in alphabetical order. The main disadvantage is that you can't organise components into files as naturally as you might want.

Older versions of roxygen2 required explicit `@usage`, `@alias` and `@docType` tags for document S4 objects. However, as of version 3.0.0, roxygen2 generates the correct values automatically so you no longer need to use them. If you're upgrading from an old version, you can delete these tags.

5.7.3 RC

Reference classes are different to S3 and S4 because methods are associated with classes, not generics. RC also has a special convention for documenting methods: the **docstring**. The docstring is a string placed inside the definition of the method which briefly describes what it does. This makes documenting RC simpler than S4 because you only need one roxygen block per class.

```r
#' A Reference Class to represent a bank account.
#'
#' @field balance A length-one numeric vector.
Account <- setRefClass("Account",
  fields = list(balance = "numeric"),
  methods = list(
    withdraw = function(x) {
      "Withdraw money from account. Allows overdrafts"
      balance <<- balance - x
    }
  )
)
```

Methods with doc strings will be included in the “Methods” section of the class documentation. Each documented method will be listed with an automatically generated usage statement and its doc string. Also note the use of `@field` instead of `@slot`.

5.8 Special characters

There are three special characters that need special handling if you want them to appear in the final documentation:

- `@`, which usually marks the start of a roxygen tag. Use `@@` to insert a literal `@` in the final documentation.
- `%`, which usually marks the start of a latex comment which continues to the end of the line. Use `\%` to insert a literal `%` in the output document. The escape is not needed in examples.
- `\`, which usually marks the start of a latex escaping. Use `\\` to insert a literal `\` in the documentation.

5.9 Do repeat yourself

There is a tension between the DRY (don't repeat yourself) principle of programming and the need for documentation to be self-contained. It's frustrating to have to navigate through multiple help files in order to pull together all the pieces you need. Roxygen2 provides two ways to avoid repetition in the source, while still assembling everything into one documentation file:

- The ability to reuse parameter documentation with `@inheritParams`.
- The ability to document multiple functions in the same place with `@describeIn` or `@rdname`

5.9.1 Inheriting parameters from other functions

You can inherit parameter descriptions from other functions using `@inheritParams source_function`. This tag will bring in all documentation for parameters that are undocumented in the current function, but documented in the source function. The source can be a function in the current package, via `@inheritParams function`, or another package, via `@inheritParams package::function`. For example the following documentation:

```
#' @param a This is the first argument
foo <- function(a) a + 10

#' @param b This is the second argument
#' @inheritParams foo
bar <- function(a, b) {
  foo(a) * 10
}
```

is equivalent to

```
#' @param a This is the first argument
#' @param b This is the second argument
bar <- function(a, b) {
  foo(a) * 10
}
```

Note that inheritance does not chain. In other words, the `source_function` must always be the function that defines the parameter using `@param`.

5.9.2 Documenting multiple functions in the same file

You can document multiple functions in the same file by using either `@rdname` or `@describeIn`. However, it's a technique best used with caution: documenting too many functions in one place leads to confusing documentation. You should use it when functions have very similar arguments, or have complementary effects (e.g., `open()` and `close()` methods).

`@describeIn` is designed for the most common cases:

- Documenting methods in a generic.
- Documenting methods in a class.
- Documenting functions with the same (or similar arguments).

It generates a new section, named either "Methods (by class)", "Methods (by generic)" or "Functions". The section contains a bulleted list describing each function. They're labelled so that you know what function or method it's talking about. Here's an example, documenting an imaginary new generic:

```
#' Foo bar generic
#'
#' @param x Object to foo.
foobar <- function(x) UseMethod("foobar")

#' @describeIn foobar Difference between the mean and the median
foobar.numeric <- function(x) abs(mean(x) - median(x))

#' @describeIn foobar First and last values pasted together in a string.
foobar.character <- function(x) paste0(x[1], "-", x[length(x)])
```

An alternative to `@describeIn` is `@rdname`. It overrides the default file name generated by roxygen and merges documentation for multiple objects into one file. This gives you the complete freedom to combine documentation as you see fit.

There are two ways to use `@rdname`. You can add documentation to an existing function:

```
#' Basic arithmetic
#'
#' @param x,y numeric vectors.
add <- function(x, y) x + y

#' @rdname add
times <- function(x, y) x * y
```

Or, you can create a dummy documentation file by documenting `NULL` and setting an informative `@name`.

```
#' Basic arithmetic
#'
#' @param x,y numeric vectors.
#' @name arith
NULL
#> NULL

#' @rdname arith
add <- function(x, y) x + y

#' @rdname arith
times <- function(x, y) x * y
```

5.10 Text formatting reference sheet

Within roxygen tags, you use `.Rd` syntax to format text. This vignette shows you examples of the most important commands. The full details are described in R extensions (`http://cran.r-project.org/doc/manuals/R-exts.html#Marking-text`).

Note that `\` and `%` are special characters in the Rd format. To insert a literal `%` or `\`, escape them with a backslash `\\`, `\%`.

5.10.1 Character formatting

- `\emph{italics}`: *italics*.
- `\strong{bold}`: **bold**.
- `\code{r_function_call(with = "arguments")}`: `r_function_call(with = "arguments")` (format inline code)
- `\preformatted{}`: format text as-is, can be used for multi-line code

5.10.2 Links

To other documentation:

- `\code{\link{function}}`: function in this package.
- `\code{\link[MASS]{abbey}}`: function in another package.

- `\link[=dest]{name}`: link to dest, but show name.
- `\linkS4class{abc}`: link to an S4 class.

To the web:

- `\url{http://rstudio.com}`: a url.
- `\href{http://rstudio.com}{Rstudio}`:, a url with custom link text.
- `\email{hadley@@rstudio.com}` (note the doubled `@`): an email address.

5.10.3 Lists

- Ordered (numbered) lists:

  ```
  #' \enumerate{
  #'   \item First item
  #'   \item Second item
  #' }
  ```

- Unordered (bulleted) lists:

  ```
  #' \itemize{
  #'   \item First item
  #'   \item Second item
  #' }
  ```

- Definition (named) lists:

  ```
  #' \describe{
  #'   \item{One}{First item}
  #'   \item{Two}{Second item}
  #' }
  ```

5.10.4 Mathematics

You can use standard LaTeX math (with no extensions). Choose between either inline or block display:

- `\eqn{a + b}`: inline equation.
- `\deqn{a + b}`: display (block) equation.

5.10.5 Tables

Tables are created with `\tabular{}`. It has two arguments:

1. Column alignment, specified by letter for each column (`l` = left, `r` = right, `c` = centre.)
2. Table contents, with columns separated by `\tab` and rows by `\cr`.

The following function turns 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))

  cols <- lapply(df, format, ...)
  contents <- do.call("paste",
    c(cols, list(sep = " \\tab ", collapse = "\\cr\n  ")))

  paste("\\tabular{", paste(col_align, collapse = ""), "}{\n  ",
    contents, "\n}\n", sep = "")
}

cat(tabular(mtcars[1:5, 1:5]))
#> \tabular{rrrrr}{
#>   21.0 \tab 6 \tab 160 \tab 110 \tab 3.90\cr
#>   21.0 \tab 6 \tab 160 \tab 110 \tab 3.90\cr
#>   22.8 \tab 4 \tab 108 \tab  93 \tab 3.85\cr
#>   21.4 \tab 6 \tab 258 \tab 110 \tab 3.08\cr
#>   18.7 \tab 8 \tab 360 \tab 175 \tab 3.15
#> }
```

5.10.6 Notes

Chapter 6

Vignettes: long-form documentation

A vignette is a long-form guide to your package. Function documentation is great if you know the name of the function you need, but it's useless otherwise. A vignette is like a book chapter or an academic paper: it can describe the problem that your package is designed to solve, and then show the reader how to solve it. A vignette should divide functions into useful categories, and demonstrate how to coordinate multiple functions to solve problems. Vignettes are also useful if you want to explain the details of your package. For example, if you have implemented a complex statistical algorithm, you might want to describe all the details in a vignette so that users of your package can understand what's going on under the hood, and be confident that you've implemented the algorithm correctly.

Many existing packages have vignettes. You can see all the installed vignettes with `browseVignettes()`. To see the vignette for a specific package, use the argument, `browseVignettes("packagename")`. Each vignette provides three things: the original source file, a readable HTML page or PDF, and a file of R code. You can read a specific vignette with `vignette(x)`, and see its code with `edit(vignette(x))`. To see vignettes for a package you haven't installed, look at its CRAN page, e.g., `http://cran.r-project.org/web/packages/dplyr`.

Before R 3.0.0, the only way to create a vignette was with Sweave. This was challenging because Sweave only worked with LaTeX, and LaTeX is both hard to learn and slow to compile. Now, any package can provide a vignette **engine**, a standard interface for turning input files into HTML or PDF vignettes. In this chapter, we're going to use the R markdown vignette engine provided by knitr (`http://yihui.name/knitr/`). I recommend this engine because:

- You write in Markdown, a plain text formatting system. Markdown is limited compared to LaTeX, but this limitation is good because it forces you to focus on the content.

- It can intermingle text, code and results (both textual and visual).
- Your life is further simplified by the rmarkdown package (`http://rmarkdown.rstudio.com/`), which coordinates Markdown and knitr by using pandoc (`http://johnmacfarlane.net/pandoc`) to convert Markdown to HTML and by providing many useful templates.

Switching from Sweave to R Markdown had a profound impact on my use of vignettes. Previously, making a vignette was painful and slow and I rarely did it. Now, vignettes are an essential part of my packages. I use them whenever I need to explain a complex topic, or to show how to solve a problem with multiple steps.

Currently, the easiest way to get R Markdown is to use RStudio (`http://www.rstudio.com/products/rstudio/download/preview/`). RStudio will automatically install all of the needed prerequisites. If you don't use RStudio, you'll need to:

1. Install the rmarkdown package with `install.packages("rmarkdown")`.
2. Install pandoc (`http://johnmacfarlane.net/pandoc/installing.html`).

6.1 Vignette workflow

To create your first vignette, run:

```r
devtools::use_vignette("my-vignette")
```

This will:

1. Create a `vignettes/` directory.
2. Add the necessary dependencies to `DESCRIPTION` (i.e. it adds knitr to the `Suggests` and `VignetteBuilder` fields).
3. Draft a vignette, `vignettes/my-vignette.Rmd`.

The draft vignette has been designed to remind you of the important parts of an R Markdown file. It serves as a useful reference when you're creating a new vignette.

Once you have this file, the workflow is straightforward:

1. Modify the vignette.

2. Press Ctrl/Cmd + Shift + K (or click Knit) to knit the vignette and preview the output.

There are three important components to an R Markdown vignette:

- The initial metadata block.
- Markdown for formatting text.
- Knitr for intermingling text, code and results.

These are described in the following sections.

6.2 Metadata

The first few lines of the vignette contain important metadata. The default template contains the following information:

```
---
title: "Vignette Title"
author: "Vignette Author"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Vignette Title}
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---
```

This metadata is written in yaml (`http://www.yaml.org/`), a format designed to be both human and computer readable. The basics of the syntax is much like the `DESCRIPTION` file, where each line consists of a field name, a colon, then the value of the field. The one special YAML feature we're using here is `>`. It indicates the following lines of text are plain text and shouldn't use any special YAML features.

The fields are:

- Title, author and date: this is where you put the vignette's title, author and date. You'll want to fill these in yourself (you can delete them if you don't want the title block at the top of the page). The date is filled in by default: it uses a special knitr syntax (explained below) to insert today's date.

- Output: this tells rmarkdown which output formatter to use. There are many options that are useful for regular reports (including html, pdf, slideshows, …) but `rmarkdown::html_vignette` has been specifically designed to work well inside packages. See `?rmarkdown::html_vignette` for more details.

- Vignette: this contains a special block of metadata needed by R. Here, you can see the legacy of LaTeX vignettes: the metadata looks like LaTeX commands. You'll need to modifiy the `\VignetteIndexEntry` to provide the title of your vignette as you'd like it to appear in the vignette index. Leave the other two lines as is. They tell R to use `knitr` to process the file, and that the file is encoded in UTF-8 (the only encoding you should ever use to write vignettes).

6.3 Markdown

R Markdown vignettes are written in Markdown, a light weight markup language. John Gruber, the author of Markdown, summarises the goals and philosophy of Markdown:

> Markdown is intended to be as easy-to-read and easy-to-write as is feasible.
>
> Readability, however, is emphasized above all else. A Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. While Markdown's syntax has been influenced by several existing text-to-HTML filters — including Setext, atx, Textile, reStructuredText, Grutatext, and EtText — the single biggest source of inspiration for Markdown's syntax is the format of plain text email.
>
> To this end, Markdown's syntax is comprised entirely of punctuation characters, which punctuation characters have been carefully chosen so as to look like what they mean. E.g., asterisks around a word actually look like *emphasis*. Markdown lists look like, well, lists. Even blockquotes look like quoted passages of text, assuming you've ever used email.

Markdown isn't as powerful as LaTeX, reStructuredText or docbook, but it's simple, easy to write, and easy to read even when it's not rendered. I find Markdown's constraints helpful for writing because it lets me focus on the content, and prevents me from messing around with the styling.

If you've never used Markdown before, a good place to start is John Gruber's Markdown syntax documentation (`http://daringfireball.net/projects/`

markdown/syntax). Pandoc's implementation of Markdown rounds off some of the rough edges and adds a number of new features, so I also recommend familiarising yourself with the pandoc readme (http://johnmacfarlane.net/pandoc/README.html). When editing a Markdown document, RStudio presents a dropdown menu via the question mark icon, which offers a Markdown reference card.

The sections below show you what I think are the most important features of pandoc's Markdown dialect. You should be able to learn the basics in under 15 minutes.

6.3.1 Sections

Headings are identified by #:

```
# Heading 1
## Heading 2
### Heading 3
```

Create a horizontal rule with three or more hyphens (or asterisks):

```
--------
********
```

6.3.2 Lists

Basic unordered lists use *:

```
* Bulleted list
* Item 2
    * Nested bullets need a 4-space indent.
    * Item 2b
```

If you want multiparagraph lists, the second and subsequent paragraphs need additional indenting:

```
  * It's possible to put multiple paragraphs of text in a list item.

    But to do that, the second and subsequent paragraphs must be
    indented by four or more spaces. It looks better if the first
    bullet is also indented.
```

Ordered lists use: 1.:

```
1. Item 1
1. Item 2
1. Items are numbered automatically, even though they all start with 1.
```

You can intermingle ordered and bulleted lists, as long as you adhere to the four space rule:

```
1.  Item 1.
    *  Item a
    *  Item b
1.  Item 2.
```

Definition lists use :

```
Definition
  : a statement of the exact meaning of a word, especially in a dictionary.
List
  : a number of connected items or names written or printed consecutively,
    typically one below the other.
  : barriers enclosing an area for a jousting tournament.
```

6.3.3 Inline formatting

Inline format is similarly simple:

```
_italic_ or *italic*
__bold__ or **bold**
[link text](destination)
<http://this-is-a-raw-url.com>
```

6.3.4 Tables

There are four types of tables (`http://johnmacfarlane.net/pandoc/README.html#tables`). I recommend using the pipe table which looks like this:

```
| Right | Left | Default | Center |
|------:|:-----|---------|:------:|
|   12  |  12  |    12   |    12  |
|  123  |  123 |   123   |   123  |
|     1 |     1 |     1   |     1  |
```

Notice the use of the `:` in the spacer under the heading. This determines the alignment of the column.

If the data underlying your table exists in R, don't lay it out by hand. Instead use `knitr::kable()`, or look at printr (`https://github.com/yihui/printr`) or pander (`http://rapporter.github.io/pander/`).

6.3.5 Code

For inline code use `` `code` ``.

For bigger blocks of code, use ```` ``` ````. These are known as "fenced" code blocks:

~~~~
```
# A comment
add <- function(a, b) a + b
```
~~~~

To add syntax highlighting to the code, put the language name after the backtick:

~~~~
```c
int add(int a, int b) {
  return a + b;
}
```
~~~~

(At time of printing, languages supported by pandoc were: actionscript, ada, apache, asn1, asp, awk, bash, bibtex, boo, c, changelog, clojure, cmake, coffee, coldfusion, commonlisp, cpp, cs, css, curry, d, diff, djangotemplate, doxygen, doxygenlua, dtd, eiffel, email, erlang, fortran, fsharp, gnuassembler, go, haskell, haxe, html, ini, java, javadoc, javascript, json, jsp, julia, latex, lex, literatecurry, literatehaskell, lua, makefile, mandoc, matlab, maxima, metafont, mips, modula2, modula3, monobasic, nasm, noweb, objectivec, objectivecpp, ocaml, octave, pascal, perl, php, pike, postscript, prolog, python, r, relaxngcompact, rhtml, ruby, rust, scala, scheme, sci, sed, sgml, sql, sqlmysql, sqlpostgresql, tcl, texinfo, verilog, vhdl, xml, xorg, xslt, xul, yacc, yaml. Syntax highlighting is done by the haskell package highlighting-kate (`http://johnmacfarlane.net/highlighting-kate`); see the website for current list.)

When you include R code in your vignette, you usually won't use ```` ```r ````. Instead, you'll use ```` ```{r} ````, which is specially processed by knitr, as described next.

6.4 Knitr

Knitr allows you to intermingle code, results and text. Knitr takes R code, runs it, captures the output, and translates it into formatted Markdown. Knitr captures all printed output, messsages, warnings, errors (optionally) and plots (basic graphics, lattice & ggplot and more).

Consider the simple example below. Note that a knitr block looks similar to a fenced code block, but instead of using `r`, you using `{r}`.

~~~
```{r}
# Add two numbers together
add <- function(a, b) a + b
add(10, 20)
```
~~~

This generates the following Markdown:

~~~
```r
# Add two numbers together
add <- function(a, b) a + b
add(10, 20)
## [1] 30
```
~~~

Which, in turn, is rendered as:

```r
# Add two numbers together
add <- function(a, b) a + b
add(10, 20)
## 30
```

Once you start using knitr, you'll never look back. Because your code is always run when you build the vignette, you can rest assured knowing that all your code works. There's no way for your input and output to be out of sync.

6.4.1 Options

You can specify additional options to control the rendering:

- To affect a single block, add the block settings:

~~~
```{r, opt1 = val1, opt2 = val2}
# code
```
~~~

- To affect all blocks, call `knitr::opts_chunk$set()` in a knitr block:

  ~~~
  ```{r, echo = FALSE}
  knitr::opts_chunk$set(
    opt1 = val1,
    opt2 = val2
  )
  ```
  ~~~

The most important options are:

- `eval = FALSE` prevents evaluation of the code. This is useful if you want to show some code that would take a long time to run. Be careful when you use this: since the code is not run, it's easy to introduce bugs. (Also, your users will be puzzled when they copy & paste code and it doesn't work.)

- `echo = FALSE` turns off the printing of the code *input* (the output will still be printed). Generally, you shouldn't use this in vignettes because understanding what the code is doing is important. It's more useful when writing reports since the code is typically less important than the output.

- `results = "hide"` turns off the printing of code *output*.

- `warning = FALSE` and `message = FALSE` suppress the display of warnings and messages.

- `error = TRUE` captures any errors in the block and shows them inline. This is useful if you want to demonstrate what happens if code throws an error. Whenever you use `error = TRUE`, you also need to use `purl = FALSE`. This is because every vignette is accompanied by a file code that contains all the code from the vignette. R must be able to source that file without errors, and `purl = FALSE` prevents the code from being inserted into that document.

- `collapse = TRUE` and `comment = "#>"` are my preferred way of displaying code output. I usually set these globally by putting the following knitr block at the start of my document.

  ~~~
  ```{r, echo = FALSE}
  knitr::opts_chunk$set(collapse = TRUE, comment = "#>")
  ```
  ~~~

- `results = "asis"` treats the output of your R code as literal Markdown. This is useful if you want to generate text from your R code. For example, if you want to generate a table using the pander package, you'd do:

  ~~~
  ```{r, results = "asis"}
  pander::pandoc.table(iris[1:3, 1:4])
  ```
  ~~~

  That generates a Markdown table that looks like:

  ```
  -------------------------------------------------------
   Sepal.Length   Sepal.Width   Petal.Length   Petal.Width
  -------------- ------------- -------------- -------------
       5.1            3.5           1.4            0.2

       4.9             3            1.4            0.2

       4.7            3.2           1.3            0.2
  ---------------------------------------------------------
  ```

  Which makes a table that looks like:

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
5.1	3.5	1.4	0.2
4.9	3	1.4	0.2
4.7	3.2	1.3	0.2

- `fig.show = "hold"` holds all figures until the end of the code block.

- `fig.width = 5` and `fig.height = 5` set the height and width of figures (in inches).

Many other options are described at `http://yihui.name/knitr/options`.

6.5 Development cycle

Run code a chunk at a time using Cmd + Alt + C. Re-run the entire document in a fresh R session using Knit (Ctrl/Cmd + Shift + K).

You can build all vignettes from the console with `devtools::build_vignettes()`, but this is rarely useful. Instead use `devtools::build()` to create a package bundle with the vignettes included. RStudio's "Build & reload" does not build vignettes to save time. Similarly, `devtools::install_github()` (and friends) will not build vignettes by default because they're time consuming and may require additional packages. You can force building with `devtools::install_github(build_vignettes = TRUE)`. This will also install all suggested packages.

6.6 Advice for writing vignettes

> If you're thinking without writing, you only think you're thinking.
> — Leslie Lamport

When writing a vignette, you're teaching someone how to use your package. You need to put yourself in the readers' shoes, and adopt a "beginner's mind". This can be difficult because it's hard to forget all of the knowledge that you've already internalised. For this reason, I find teaching in-person a really useful way to get feedback on my vignettes. Not only do you get that feedback straight away but it's also a much easier way to learn what people already know.

A useful side effect of this approach is that it helps you improve your code. It forces you to re-see the initial onboarding process and to appreciate the parts that are hard. Every time that I've written text that describes the initial experience, I've realised that I've missed some important functions. Adding those functions not only helps my users, but it often also helps me! (This is one of the reasons that I like writing books).

- I strongly recommend literally anything written by Kathy Sierra. Her old blog, Creating passionate users (`http://headrush.typepad.com/`) is full of advice about programming, teaching, and how to create valuable tools. I thoroughly recommend reading through all the older content. Her new blog, Serious Pony (`http://seriouspony.com/blog/`), doesn't have as much content, but it has some great articles.

- If you'd like to learn how to write better, I highly recommend Style: Lessons in Clarity and Grace (`http://amzn.com/0321898680`) by Joseph M. Williams and Joseph Bizup. It helps you understand the structure of writing so that you'll be better able to recognise and fix bad writing.

Writing a vignette also makes a nice break from coding. In my experience, writing uses a different part of the brain from programming, so if you're sick of programming, try writing for a bit. (This is related to the idea of structured programming (`http://www.structuredprocrastination.com/`).).

6.6.1 Organisation

For simpler packages, one vignette is often sufficient. But for more complicated packages you may actually need more than one. In fact, you can have as many vignettes as you like. I tend to think of them like chapters of a book - they should be self-contained, but still link together into a cohesive whole.

Although it's a slight hack, you can link various vignettes by taking advantage of how files are stored on disk: to link to vignette `abc.Rmd`, just make a link to `abc.html`.

6.7 CRAN notes

Note that since you build vignettes locally, CRAN only receives the html/pdf and the source code. However, CRAN does not re-build the vignette. It only checks that the code is runnable (by running it). This means that any packages used by the vignette must be declared in the `DESCRIPTION`. But this also means that you can use Rmarkdown (which uses pandoc) even though CRAN doesn't have pandoc installed.

Common problems:

- The vignette builds interactively, but when checking, it fails with an error about a missing package that you know is installed. This means that you've forgotten to declare that dependency in the `DESCRIPTION` (usually it should go in `Suggests`).

- Everything works interactively, but the vignette doesn't show up after you've installed the package. One of the following may have occurred. First, because RStudio's "build and reload" doesn't build vignettes, you may need to run `devtools::install()` instead. Next check:

    1. The directory is called `vignettes/` and not `vignette/`.
    2. Check that you haven't inadvertently excluded the vignettes with `.Rbuildignore`
    3. Ensure you have the necessary vignette metadata.

- If you use `error = TRUE`, you must use `purl = FALSE`.

You'll need to watch the file size. If you include a lot of graphics, it's easy to create a very large file. There are no hard and fast rules, but if you have a very large vignette be prepared to either justify the file size, or to make it smaller.

6.8 Where next

If you'd like more control over the appearance of your vignette, you'll need to learn more about Rmarkdown. The website, `http://rmarkdown.rstudio.com`, is a great place to start. There you can learn about alternative output formats (like LaTex and pdf) and how you can incorporate raw HTML and LateX if you need additional control.

If you write a nice vignette, consider submitting it to the *Journal of Statistical Software* or *The R Journal*. Both journals are electronic only and peer-reviewed. Comments from reviewers can be very helpful for improving the quality of your vignette and the related software.

Chapter 7

Testing

Testing is a vital part of package development. It ensures that your code does what you want it to do. Testing, however, adds an additional step to your development workflow. The goal of this chapter is to show you how to make this task easier and more effective by doing formal automated testing using the testthat package.

Up until now, your workflow probably looks like this:

1. Write a function.
2. Load it with Ctrl/Cmd + Shift + L or `devtools::load_all()`.
3. Experiment with it in the console to see if it works.
4. Rinse and repeat.

While you *are* testing your code in this workflow, you're only doing it informally. The problem with this approach is that when you come back to this code in 3 months time to add a new feature, you've probably forgotten some of the informal tests you ran the first time around. This makes it very easy to break code that used to work.

I started using automated tests because I discovered I was spending too much time re-fixing bugs that I'd already fixed before. While writing code or fixing bugs, I'd perform interactive tests to make sure the code worked. But I never had a system which could store those tests so I could re-run them as needed. I think that this is a common practice among R programmers. It's not that you don't test your code, it's that you don't automate your tests.

In this chapter you'll learn how to graduate from using informal ad hoc testing, done at the command line, to formal automated testing (aka unit testing). While turning casual interactive tests into reproducible scripts requires a little more work up front, it pays off in four ways:

- Fewer bugs. Because you're explicit about how your code should behave you will have fewer bugs. The reason why is a bit like the reason double entry book-keeping works: because you describe the behaviour of your code in two places, both in your code and in your tests, you are able to check one against the other. By following this approach to testing, you can be sure that bugs that you've fixed in the past will never come back to haunt you.
- Better code structure. Code that's easy to test is usually better designed. This is because writing tests forces you to break up complicated parts of your code into separate functions that can work in isolation. This reduces duplication in your code. As a result, functions will be easier to test, understand and work with (it'll be easier to combine them in new ways).
- Easier restarts. If you always finish a coding session by creating a failing test (e.g. for the next feature you want to implement), testing makes it easier for you to pick up where you left off: your tests will let you know what to do next.
- Robust code. If you know that all the major functionality of your package has an associated test, you can confidently make big changes without worrying about accidentally breaking something. For me, this is particularly useful when I think I have a simpler way to accomplish a task (usually the reason my solution is simpler is that I've forgotten an important use case!).

If you're familiar with unit testing in other languages, you should note that there are some fundamental differences with testthat. This is because R is, at heart, more a functional programming language than an object oriented programming language. For instance, because R's main OO systems (S3 and S4) are based on generic functions (i.e., methods belong to functions not classes), testing approaches built around objects and methods don't make much sense.

7.1 Test workflow

To set up your package to use testthat, run:

```
devtools::use_testthat()
```

This will:

1. Create a `tests/testthat` directory.
2. Adds testthat to the `Suggests` field in the `DESCRIPTION`.

3. Creates a file `tests/testthat.R` that runs all your tests are when `R CMD check` runs. (You'll learn more about that in Chapter 14.)

Once you're set up the workflow is simple:

1. Modify your code or tests.
2. Test your package with Ctrl/Cmd + Shift + T or `devtools::test()`.
3. Repeat until all tests pass.

The testing output looks like this:

```
Expectation : ...........
rv : ...
Variance : ....123.45.
```

Each line represents a test file. Each `.` represents a passed test. Each number represents a failed test. The numbers index into a list of failures that provides more details:

```
1. Failure(@test-variance.R#22): Variance correct for discrete uniform rvs -----
VAR(dunif(0, 10)) not equal to var_dunif(0, 10)
Mean relative difference: 3

2. Failure(@test-variance.R#23): Variance correct for discrete uniform rvs -----
VAR(dunif(0, 100)) not equal to var_dunif(0, 100)
Mean relative difference: 3.882353
```

Each failure gives a description of the test (e.g., "Variance correct for discrete uniform rvs"), its location (e.g., "@test-variance.R#22"), and the reason for the failure (e.g., "VAR(dunif(0, 10)) not equal to var_dunif(0, 10)"). The goal is to pass all the tests.

7.2 Test structure

A test file lives in `tests/testthat/`. Its name must start with `test`. Here's an example of a test file from the stringr package:

```
library(stringr)
context("String length")

test_that("str_length is number of characters", {
  expect_equal(str_length("a"), 1)
  expect_equal(str_length("ab"), 2)
  expect_equal(str_length("abc"), 3)
})

test_that("str_length of factor is length of level", {
  expect_equal(str_length(factor("a")), 1)
  expect_equal(str_length(factor("ab")), 2)
  expect_equal(str_length(factor("abc")), 3)
})

test_that("str_length of missing is missing", {
  expect_equal(str_length(NA), NA_integer_)
  expect_equal(str_length(c(NA, 1)), c(NA, 1))
  expect_equal(str_length("NA"), 2)
})
```

Tests are organised hierarchically: **expectations** are grouped into **tests** which are organised in **files**:

- An **expectation** is the atom of testing. It describes the expected result of a computation: Does it have the right value and right class? Does it produce error messages when it should? An expectation automates visual checking of results in the console. Expectations are functions that start with `expect_`.

- A **test** groups together multiple expectations to test the output from a simple function, a range of possibilities for a single parameter from a more complicated function, or tightly related functionality from across multiple functions. This is why they are sometimes called **unit** as they test one unit of functionality. A test is created with `test_that()` .

- A **file** groups together multiple related tests. Files are given a human readable name with `context()`.

These are described in detail below.

7.2.1 Expectations

An expectation is the finest level of testing. It makes a binary assertion about whether or not a function call does what you expect. All expectations have a similar structure:

- They start with `expect_`.

- They have two arguments: the first is the actual result, the second is what you expect.
- If the actual and expected results don't agree, testthat throws an error.

While you'll normally put expectations inside tests inside files, you can also run them directly. This makes it easy to explore expectations interactively. There are almost 20 expectations in the testthat package. The most important are discussed below.

- There are two basic ways to test for equality: `expect_equal()`, and `expect_identical()`. `expect_equal()` is the most commonly used: it uses `all.equal()` to check for equality within a numerical tolerance:

  ```
  expect_equal(10, 10)
  expect_equal(10, 10 + 1e-7)
  expect_equal(10, 11)
  #> Error: 10 not equal to 11
  #> 11 - 10 == 1
  ```

  If you want to test for exact equivalence, or need to compare a more exotic object like an environment, use `expect_identical()`. It's built on top of `identical()`:

  ```
  expect_equal(10, 10 + 1e-7)
  expect_identical(10, 10 + 1e-7)
  #> Error: 10 is not identical to 10 + 1e-07. Differences:
  #> Objects equal but not identical
  ```

- `expect_match()` matches a character vector against a regular expression. The optional `all` argument controls whether all elements or just one element needs to match. This is powered by `grepl()` (additional arguments like `ignore.case = FALSE` or `fixed = TRUE` are passed on down).

  ```
  string <- "Testing is fun!"

  expect_match(string, "Testing")
  # Fails, match is case-sensitive
  expect_match(string, "testing")
  #> Error: string does not match 'testing'. Actual value: "Testing is fun!"

  # Additional arguments are passed to grepl:
  expect_match(string, "testing", ignore.case = TRUE)
  ```

- Four variations of `expect_match()` let you check for other types of result: `expect_output()`, inspects printed output; `expect_message()`, messages; `expect_warning()`, warnings; and `expect_error()` errors.

```
a <- list(1:10, letters)

expect_output(str(a), "List of 2")
expect_output(str(a), "int [1:10]", fixed = TRUE)

expect_message(library(mgcv), "This is mgcv")
```

With `expect_message()`, `expect_warning()`, `expect_error()` you can leave the second argument blank if you just want to see if a message, warning or error is created. However, it's normally better to be explicit, and provide some text from the message.

```
expect_warning(log(-1))
expect_error(1 / "a")

# But always better to be explicit
expect_warning(log(-1), "NaNs produced")
expect_error(1 / "a", "non-numeric argument")

# Failure to produce a warning or error when expected is an error
expect_warning(log(0))
#> Error: log(0) no warnings given
expect_error(1 / 2)
#> Error: 1/2 code did not generate an error
```

- `expect_is()` checks that an object `inherit()`s from a specified class.

```
model <- lm(mpg ~ wt, data = mtcars)
expect_is(model, "lm")
expect_is(model, "glm")
#> Error: model inherits from lm not glm
```

- `expect_true()` and `expect_false()` are useful catchalls if none of the other expectations do what you need.

- Sometimes you don't know exactly what the result should be, or it's too complicated to easily recreate in code. In that case the best you can do is check that the result is the same as last time. `expect_equal_to_reference()` caches the result the first time its run, and then compares it to subsequent runs. If for some reason the result does change, just delete the cache (*) file and re-test.

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. You don't know the goal of the expectation. Tests, described next, organise expectations into coherent blocks that describe the overall goal of a set of expectations.

7.3 Writing tests

Each test should have an informative name and cover a single unit of functionality. The idea is that when a test fails, you'll know what's wrong and where in your code to look for the problem. You create a new test using `test_that()`, with test name and code block as arguments. The test name should complete the sentence "Test that …". The code block should be a collection of expectations.

It's up to you how to organise your expectations into tests. The main thing is that the message associated with the test should be informative so that you can quickly narrow down the source of the problem. Try to avoid putting too many expectations in one test - it's better to have more smaller tests than fewer larger tests.

Each test is run in its own environment and is self-contained. However, testthat doesn't know how to cleanup after actions affect the R landscape:

- The filesystem: creating and deleting files, changing the working directory, etc.
- The search path: `library()`, `attach()`.
- Global options, like `options()` and `par()`.

When you use these actions in tests, you'll need to clean up after yourself. While 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 you can 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.

7.3.1 What to test

> Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead. — Martin Fowler

There is a fine balance to writing tests. Each test that you write makes your code less likely to change inadvertently; but it also can make it harder to change your code on purpose. It's hard to give good general advice about writing tests, but you might find these points helpful:

- Focus on testing the external interface to your functions - if you test the internal interface, then it's harder to change the implementation in the future because as well as modifying the code, you'll also need to update all the tests.

- Strive to test each behaviour in one and only one test. Then if that behaviour later changes you only need to update a single test.
- Avoid testing simple code that you're confident will work. Instead focus your time on code that you're not sure about, is fragile, or has complicated interdependencies. That said, I often find I make the most mistakes when I falsely assume that the problem is simple and doesn't need any tests.
- Always write a test when you discover a bug. You may find it helpful to adopt the test-first philosphy. There you always start by writing the tests, and then write the code that makes them pass. This reflects an important problem solving strategy: start by establishing your success critieria, how you know if you've solved the problem.

7.3.2 Skipping a test

Sometimes it's impossible to perform a test - you may not have an internet connection or you may be missing an important file. Unfortunately, another likely reason follows from this simple rule: the more machines you use to write your code, the more likely it is that you won't be able to run all of your tests. In short, there are times when, instead of getting a failure, you just want to skip a test. To do that, you can use the `skip()` function - rather than throwing an error it simply prints an `S` in the output.

```
check_api <- function() {
  if (not_working()) {
    skip("API not available")
  }
}

test_that("foo api returns bar when given baz", {
  check_api()
  ...
})
```

7.3.3 Building your own testing tools

As you start to write more tests, you might notice duplication in your code. For example, the following code shows one test of the `floor_date()` function from `library(lubridate)`. There are seven expectations that check the results of rounding a date down to the nearest second, minute, hour, etc. There's a lot of duplication (which increases the chance of bugs), so we might want to extract common behaviour into a new function.

```
library(lubridate)
test_that("floor_date works for different units", {
  base <- as.POSIXct("2009-08-03 12:01:59.23", tz = "UTC")

  expect_equal(floor_date(base, "second"),
    as.POSIXct("2009-08-03 12:01:59", tz = "UTC"))
  expect_equal(floor_date(base, "minute"),
    as.POSIXct("2009-08-03 12:01:00", tz = "UTC"))
  expect_equal(floor_date(base, "hour"),
    as.POSIXct("2009-08-03 12:00:00", tz = "UTC"))
  expect_equal(floor_date(base, "day"),
    as.POSIXct("2009-08-03 00:00:00", tz = "UTC"))
  expect_equal(floor_date(base, "week"),
    as.POSIXct("2009-08-02 00:00:00", tz = "UTC"))
  expect_equal(floor_date(base, "month"),
    as.POSIXct("2009-08-01 00:00:00", tz = "UTC"))
  expect_equal(floor_date(base, "year"),
    as.POSIXct("2009-01-01 00:00:00", tz = "UTC"))
})
```

I'd start by defining a couple of helper functions to make each expectation more concise. That allows each test to fit on one line, so you can line up actual and expected values to make it easier to see the differences:

```
test_that("floor_date works for different units", {
  base <- as.POSIXct("2009-08-03 12:01:59.23", tz = "UTC")
  floor_base <- function(unit) floor_date(base, unit)
  as_time <- function(x) as.POSIXct(x, tz = "UTC")

  expect_equal(floor_base("second"), as_time("2009-08-03 12:01:59"))
  expect_equal(floor_base("minute"), as_time("2009-08-03 12:01:00"))
  expect_equal(floor_base("hour"),   as_time("2009-08-03 12:00:00"))
  expect_equal(floor_base("day"),    as_time("2009-08-03 00:00:00"))
  expect_equal(floor_base("week"),   as_time("2009-08-02 00:00:00"))
  expect_equal(floor_base("month"),  as_time("2009-08-01 00:00:00"))
  expect_equal(floor_base("year"),   as_time("2009-01-01 00:00:00"))
})
```

We could go a step further and create a custom expectation function:

```
base <- as.POSIXct("2009-08-03 12:01:59.23", tz = "UTC")

expect_floor_equal <- function(unit, time) {
  expect_equal(floor_date(base, unit), as.POSIXct(time, tz = "UTC"))
}
expect_floor_equal("year", "2009-01-01 00:00:00")
```

However, if the expectation fails this doesn't give very informative output:

```
expect_floor_equal("year", "2008-01-01 00:00:00")
#> Error: floor_date(base, unit) not equal to as.POSIXct(time, tz = "UTC")
#> Mean absolute difference: 31622400
```

Instead you can use a little non-standard evaluation (`http://adv-r.had.co.nz/Computing-on-the-language.html`) to produce something more informative. The key is to use `bquote()` and `eval()`. In the `bquote()` call below, note the use of `.(x)` - the contents of `()` will be inserted into the call.

```
expect_floor_equal <- function(unit, time) {
  as_time <- function(x) as.POSIXct(x, tz = "UTC")
  eval(bquote(expect_equal(floor_date(base, .(unit)), as_time(.(time)))))
}
expect_floor_equal("year", "2008-01-01 00:00:00")
#> Error: floor_date(base, "year") not equal to as_time("2008-01-01 00:00:00")
#> Mean absolute difference: 31622400
```

This sort of refactoring is often worthwhile because removing redundant code makes it easier to see what's changing. Readable tests give you more confidence that they're correct.

```
test_that("floor_date works for different units", {
  as_time <- function(x) as.POSIXct(x, tz = "UTC")
  expect_floor_equal <- function(unit, time) {
    eval(bquote(expect_equal(floor_date(base, .(unit)), as_time(.(time)))))
  }

  base <- as_time("2009-08-03 12:01:59.23")
  expect_floor_equal("second", "2009-08-03 12:01:59")
  expect_floor_equal("minute", "2009-08-03 12:01:00")
  expect_floor_equal("hour",   "2009-08-03 12:00:00")
  expect_floor_equal("day",    "2009-08-03 00:00:00")
  expect_floor_equal("week",   "2009-08-02 00:00:00")
  expect_floor_equal("month",  "2009-08-01 00:00:00")
  expect_floor_equal("year",   "2009-01-01 00:00:00")
})
```

7.4 Test files

The highest-level structure of tests is the file. Each file should contain a single `context()` call that provides a brief description of its contents. Just like the files in the `R/` directory, you are free to organise your tests any way that you like. But again, the two extremes are clearly bad (all tests in one file, one file per test). You need to find a happy medium that works for you. A good starting place is to have one file of tests for each complicated function.

7.5 CRAN notes

CRAN will run your tests on all CRAN platforms: Windows, Mac, Linux and Solaris. There are a few things to bear in mind:

- Tests need to run relatively quickly - aim for under a minute. Place `skip_on_cran()` at the beginning of long-running tests that shouldn't be run on CRAN - they'll still be run locally, but not on CRAN.
- Note that tests are always run in the English language (`LANGUAGE=EN`) and with C sort order (`LC_COLLATE=C`). This minimises spurious differences between platforms.
- Be careful about testing things that are likely to be variable on CRAN machines. It's risky to test how long something takes (because CRAN machines are often heavily loaded) or to test parallel code (because CRAN runs multiple package tests in parallel, multiple cores will not always available). Numerical precision can also vary across platforms (it's often less precise on 32-bit versions of R) so use `expect_equal()` rather than `expect_identical()`.

Chapter 8

Namespace

The package namespace (as recorded in the `NAMESPACE` file) is one of the more confusing parts of building a package. It's a fairly advanced topic, and by-and-large, not that important if you're only developing packages for yourself. However, understanding namespaces is vital if you plan to submit your package to CRAN. This is because CRAN requires that your package plays nicely with other packages.

When you first start using namespaces, it'll seem like a lot of work for little gain. However, having a high quality namespace helps encapsulate your package and makes it self-contained. This ensures that other packages won't interfere with your code, that your code won't interfere with other packages, and that your package works regardless of the environment in which it's run.

8.1 Motivation

As the name suggests, namespaces provide "spaces" for "names". They provide a context for looking up the value of an object associated with a name.

Without knowing it, you've probably already used namespaces. For example, have you ever used the `::` operator? It disambiguates functions with the same name. For example, both plyr and Hmisc provide a `summarize()` function. If you load plyr, then Hmisc, `summarize()` will refer to the Hmisc version. But if you load the packages in the opposite order, `summarize()` will refer to the plyr version. This can be confusing. Instead, you can explicitly refer to specific functions: `Hmisc::summarize()` and `plyr::summarize()`. Then the order in which the packages are loaded won't matter.

Namespaces make your packages self-contained in two ways: the **imports** and the **exports**. The **imports** defines how a function in one package finds a function in another. To illustrate, consider what happens when someone changes the definition of a function that you rely on: for example, the simple `nrow()` function in base R:

```
nrow
#> function (x)
#> dim(x)[1L]
#> <bytecode: 0xc1b2c0>
#> <environment: namespace:base>
```

It's defined in terms of `dim()`. So what will happen if we override `dim()` with our own definition? Does `nrow()` break?

```
dim <- function(x) c(1, 1)
dim(mtcars)
#> [1] 1 1
nrow(mtcars)
#> [1] 32
```

Surprisingly, it does not! That's because when `nrow()` looks for an object called `dim()`, it uses the package namespace, so it finds `dim()` in the base environment, not the `dim()` we created in the global environment.

The **exports** helps you avoid conflicts with other packages by specifying which functions are available outside of your package (internal functions are available only within your package and can't easily be used by another package). Generally, you want to export a minimal set of functions; the fewer you export, the smaller the chance of a conflict. While conflicts aren't the end of the world because you can always use `::` to disambiguate, they're best avoided where possible because it makes the lives of your users easier.)

8.2 Search path

To understand why namespaces are important, you need a solid understanding of search paths. To call a function, R first has to find it. R does this by first looking in the global environment. If R doesn't find it there, it looks in the search path, the list of all the packages you have **attached**. You can see this list by running `search()`. For example, here's the search path for the code in this book:

```
search()
#>  [1] ".GlobalEnv"        "package:bookdown"
#>  [3] "package:stats"     "package:graphics"
#>  [5] "package:grDevices" "package:utils"
#>  [7] "package:datasets"  "package:methods"
#>  [9] "Autoloads"         "package:base"
```

There's an important difference between loading and attaching a package. Normally when you talk about loading a package you think of `library()`, but that actually attaches the package.

If a package is installed,

- **Loading** will load code, data and any DLLs; register S3 and S4 methods; and run the `.onLoad()` function. After loading, the package is available in memory, but because it's not in the search path, you won't be able to access its components without using `::`. Confusingly, `::` will also load a package automatically if it isn't already loaded. It's rare to load a package explicitly, but you can do so with `requireNamespace()` or `loadNamespace()`.

- **Attaching** puts the package in the search path. You can't attach a package without first loading it, so both `library()` or `require()`, load then attach the package. You can see the currently attached packages with `search()`.

If a package isn't installed, loading (and hence attaching) will fail with an error.

To see the differences more clearly, consider two ways of running `expect_that()` from the testthat package. If we use `library()`, testthat is attached to the search path. If we use `::`, it's not.

```
old <- search()
testthat::expect_equal(1, 1)
setdiff(search(), old)
#> character(0)
expect_true(TRUE)
#> Error in eval(expr, envir, enclos): could not find function "expect_true"

library(testthat)
expect_equal(1, 1)
setdiff(search(), old)
#> [1] "package:testthat"
expect_true(TRUE)
```

There are four functions that make a package available. They differ based on whether they load or attach, and what happens if the package is not found (i.e., throws an error or returns FALSE).

	Throws error	Returns `FALSE`
Load	`loadNamespace("x")`	`requireNamespace("x", quietly = TRUE)`
Attach	`library(x)`	`require(x, quietly = TRUE)`

Of the four, you should only ever use two:

- Use `library(x)` in data analysis scripts. It will throw an error if the package is not installed, and will terminate the script. You want to attach the package to save typing. Never use `library()` in a package.
- Use `requireNamespace(x, quietly = TRUE)` inside a package if you want a specific action (e.g. throw an error) depending on whether or not a suggested package is installed.

You never need to use `require()` (`requireNamespace()` is almost always better), or `loadNamespace()` (which is only needed for internal R code). You should never use `require()` or `library()` in a package: instead, use the `Depends` or `Imports` fields in the `DESCRIPTION`.

Now's a good time to come back to an important issue which we glossed over earlier. What's the difference between `Depends` and `Imports` in the `DESCRIPTION`? When should you use one or the other?

Listing a package in either `Depends` or `Imports` ensures that it's installed when needed. The main difference is that where `Imports` just *loads* the package, `Depends` *attaches* it. There are no other differences. The rest of the advice in this chapter applies whether or not the package is in `Depends` or `Imports`.

Unless there is a good reason otherwise, you should always list packages in `Imports` not `Depends`. That's because a good package is self-contained, and minimises changes to the global environment (including the search path). The only exception is if your package is designed to be used in conjunction with another package. For example, the analogue (`https://github.com/gavinsimpson/analogue`) package builds on top of vegan (`https://github.com/vegandevs/vegan`). It's not useful without vegan, so it has vegan in `Depends` instead of `Imports`. Similarly, ggplot2 should really `Depend` on scales, rather than `Importing` it.

Now that you understand the importance of the namespace, let's dive into the nitty gritty details. The two sides of the package namespace, imports and exports, are both described by the `NAMESPACE`. You'll learn what this file looks like in the next section. In the section after that, you'll learn the details of exporting and importing functions and other objects.

8.3 The NAMESPACE

The following code is an excerpt of the NAMESPACE file from the testthat package.

```
# Generated by roxygen2 (4.0.2): do not edit by hand
S3method(as.character,expectation)
S3method(compare,character)
export(auto_test)
export(auto_test_package)
export(colourise)
export(context)
exportClasses(ListReporter)
exportClasses(MinimalReporter)
importFrom(methods,setRefClass)
useDynLib(testthat,duplicate_)
useDynLib(testthat,reassign_function)
```

You can see that the NAMESPACE file looks a bit like R code. Each line contains a **directive**: S3method(), export(), exportClasses(), and so on. Each directive describes an R object, and says whether it's exported from this package to be used by others, or it's imported from another package to be used locally.

In total, there are eight namespace directives. Four describe exports:

- export(): export functions (including S3 and S4 generics).
- exportPattern(): export all functions that match a pattern.
- exportClasses(), exportMethods(): export S4 classes and methods.
- S3method(): export S3 methods.

And four describe imports:

- import(): import all functions from a package.
- importFrom(): import selected functions (including S4 generics).
- importClassesFrom(), importMethodsFrom(): import S4 classes and methods.
- useDynLib(): import a function from C. This is described in more detail in Chapter 10.

I don't recommend writing these directives by hand. Instead, in this chapter you'll learn how to generate the NAMESPACE file with roxygen2. There are three main advantages to using roxygen2:

- Namespace definitions live next to its associated function, so when you read the code it's easier to see what's being imported and exported.

- Roxygen2 abstracts away some of the details of NAMESPACE. You only need to learn one tag, @export, which will automatically generate the right directive for functions, S3 methods, S4 methods and S4 classes.
- Roxygen2 makes NAMESPACE tidy. No matter how many times you use @importFrom foo bar you'll only get one importFrom(foo, bar) in your NAMESPACE. This makes it easy to attach import directives to every function that need them, rather than trying to manage in one central place.

Note that you can choose to use roxygen2 to generate just NAMESPACE, just man/*.Rd, or both. If you don't use any namespace related tags, roxygen2 won't touch NAMESPACE. If you don't use any documentation related tags, roxygen2 won't touch man/.

8.4 Workflow

Generating the namespace with roxygen2 is just like generating function documentation with roxygen2. You use roxygen2 blocks (starting with #') and tags (starting with @). The workflow is the same:

1. Add roxygen comments to your .R files.
2. Run devtools::document() (or press Ctrl/Cmd + Shift + D in RStudio) to convert roxygen comments to .Rd files.
3. Look at NAMESPACE and run tests to check that the specification is correct.
4. Rinse and repeat until the correct functions are exported.

8.5 Exports

For a function to be usable outside of your package, you must **export** it. When you create a new package with devtools::create(), it produces a temporary NAMESPACE that exports everything in your package that doesn't start with . (a single period). If you're just working locally, it's fine to export everything in your package. However, if you're planning on sharing your package with others, it's a really good idea to only export needed functions. This reduces the chances of a conflict with another package.

To export an object, put @export in its roxygen block. For example:

```
#' @export
foo <- function(x, y, z) {
  ...
}
```

This will then generate `export()`, `exportMethods()`, `exportClass()` or `S3method()` depending on the type of the object.

You export functions that you want other people to use. Exported functions must be documented, and you must be cautious when changing their interface — other people are using them! Generally, it's better to export too little than too much. It's easy to export things that you didn't before; it's hard to stop exporting a function because it might break existing code. Always err on the side of caution, and simplicity. It's easier to give people more functionality than it is to take away stuff they're used to.

I believe that packages that have a wide audience should strive to do one thing and do it well. All functions in a package should be related to a single problem (or a set of closely related problems). Any functions not related to that purpose should not be exported. For example, most of my packages have a `utils.R` file that contains many small functions that are useful for me, but aren't part of the core purpose of those packages. I never export these functions.

```
# Defaults for NULL values
`%||%` <- function(a, b) if (is.null(a)) b else a

# Remove NULLs from a list
compact <- function(x) {
  x[!vapply(x, is.null, logical(1))]
}
```

That said, if you're creating a package for yourself, it's far less important to be so disciplined. Because you know what's in your package, it's fine to have a local "misc" package that contains a passel of functions that you find useful. But I don't think you should release such a package.

The following sections describe what you should export if you're using S3, S4 or RC.

8.5.1 S3

If you want others to be able to create instances of an S3 class, `@export` the constructor function. S3 generics are just regular R functions, you can `@export` them like functions.

S3 methods represent the most complicated case because there are four different scenarios:

- A method for an exported generic: export every method.

- A method for an internal generic: technically, you don't need to export these methods. However, I recommend exporting every S3 method you write because it's simpler and makes it less likely that you'll introduce hard to find bugs. Use `devtools::missing_s3()` to list all S3 methods that you've forgotten to export.

- A method for a generic in a required package. You'll need to import the generic (see below), and export the method.

- A method for a generic in a suggested package. Namespace directives must refer to available functions, so they can not reference suggested packages. It's possible to use package hooks and code to add this at run-time, but this is sufficiently complicated that I currently wouldn't recommend it. Instead, you'll have to design your package dependencies in a way that avoids this scenario.

8.5.2 S4

S4 classes: if you want others to be able to extend your class, `@export` it. If you want others to create instances of your class but not to extend it, `@export` the constructor function, not the class.

```
# Can extend and create with new("A", ...)
#' @export
setClass("A")

# Can extend and create with new("B", ...). You can use B()
# to construct instances in your own code, but others can not
#' @export
B <- setClass("B")

# Can create with C(...) and new("C", ...), but can't create
# a subclass that extends C
#' @export C
C <- setClass("C")

# Can extend and create with D(...) or new("D", ...)
#' @export D
#' @exportClass D
D <- setClass("D")
```

S4 generics: `@export` if you want the generic to be publicly usable.

S4 methods: you only need to `@export` methods for generics that you did not define. But I think it's a good idea to `@export` every method: that way you don't need to remember whether or not you created the generic.

8.5.3 RC

The principles used for S4 classes apply here. Note that due to the way that RC is currently implemented, it's typically impossible for your classes to be extended outside of your package.

8.5.4 Data

As you'll learn about in Chapter 9, files that live in `data/` don't use the usual namespace mechanism and don't need to be exported.

8.6 Imports

`NAMESPACE` also controls which external functions can be used by your package without having to use `::`.

It's confusing that both `DESCRIPTION` (through the `Imports` field) and `NAMESPACE` (through import directives) seem to be involved in imports. This is just an unfortunate choice of names. The `Imports` field really has nothing to do with functions imported into the namespace: it just makes sure the package is installed when your package is. It doesn't make functions available. You need to import functions in exactly the same way regardless of whether or not the package is attached.

`Depends` is just a convenience for the user: if your package is attached, it also attaches all packages listed in `Depends`. If your package is loaded, packages in `Depends` are loaded, but not attached, so you need to qualify function names with `::` or specifically import them.

It's common for packages to be listed in `Imports` in `DESCRIPTION`, but not in `NAMESPACE`. In fact, this is what I recommend: list the package in `DESCRIPTION` so that it's installed, then always refer to it explicitly with `pkg::fun()`. Unless there is a strong reason not to, it's better to be explicit. It's a little more work to write, but a lot easier to read when you come back to the code in the future. The converse is not true. Every package mentioned in `NAMESPACE` must also be present in the `Imports` or `Depends` fields.

8.6.1 R functions

If you are using just a few functions from another package, my recommendation is to note the package name in the `Imports:` field of the `DESCRIPTION` file and call the function(s) explicitly using `::`, e.g., `pkg::fun()`.

If you are using functions repeatedly, you can avoid `::` by importing the function with `@importFrom pgk fun`. This also has a small performance benefit, because `::` adds approximately 5 µs to function evaluation time.

Alternatively, if you are repeatedly using many functions from another package, you can import all of them using `@import package`. This is the least recommended solution because it makes your code harder to read (you can't tell where a function is coming from), and if you `@import` many packages, it increases the chance of conflicting function names.

8.6.2 S3

S3 generics are just functions, so the same rules for functions apply. S3 methods always accompany the generic, so as long as you can access the generic (either implicitly or explicitly), the methods will also be available. In other words, you don't need to do anything special for S3 methods. As long as you've imported the generic, all the methods will also be available.

8.6.3 S4

To use classes defined in another package, place `@importClassesFrom package ClassA ClassB ...` next to the classes that inherit from the imported classes, or next to the methods that implement a generic for the imported classes.

To use generics defined in another package, place `@importMethodsFrom package GenericA GenericB ...` next to the methods that use the imported generics.

Since S4 is implemented in the methods package, you need to make sure it's available. This is easy to overlook because while the method package is always available in the search path when you're working interactively, it's not automatically loaded by `Rscript`, the tool often used to run R from the command line.

- Pre R 3.2.0: `Depends: methods` in `DESCRIPTION`.
  Post R 3.2.0: `Imports: methods` in `DESCRIPTION`.

- Since you'll being using a lot of functions from `methods`, you'll probably also want to import the complete package with:

  ```
  #' @import methods
  NULL
  ```

  Or you might just want to import the most commonly used functions:

  ```
  #' @importFrom methods setClass setGeneric setMethod setRefClass
  NULL
  ```

Here I'm documenting "NULL" to make it clear that these directives don't apply to just one function. It doesn't matter where they go, but if you have package docs, as described in Section 5.6, that's a natural place to put them.

8.6.4 Compiled functions

To make C/C++ functions available in R, see Chapter 10.

Chapter 9

External data

It's often useful to include data in a package. If you're releasing the package to a broad audience, it's a way to provide compelling use cases for the package's functions. If you're releasing the package to a more specific audience, interested either in the data (e.g., NZ census data) or the subject (e.g., demography), it's a way to distribute that data along with its documentation (as long as your audience is R users).

There are three main ways to include data in your package, depending on what you want to do with it and who should be able to use it:

- If you want to store binary data and make it available to the user, put it in `data/`. This is the best place to put example datasets.
- If you want to store parsed data, but not make it available to the user, put it in `R/sysdata.rda`. This is the best place to put data that your functions need.
- If you want to store raw data, put it in `inst/extdata`.

A simple alternative to these three options is to include it in the source of your package, either creating by hand, or using `dput()` to serialise an existing data set into R code.

Each possible location is described in more detail below.

9.1 Exported data

The most common location for package data is (surprise!) `data/`. Each file in this directory should be a `.RData` file created by `save()` containing a single object (with the same name as the file). The easiest way to adhere to these rules is to use `devtools::use_data()`:

```
x <- sample(1000)
devtools::use_data(x, mtcars)
```

It's possible to use other types of files, but I don't recommend it because `.RData` files are already fast, small and explicit. Other options are described in `data()`. For larger datasets, you may want to experiment with the compression setting. The default is `bzip2`, but sometimes `gzip` or `xz` can create smaller files (typically at the expense of slower loading times).

If the `DESCRIPTION` contains `LazyData: true`, then datasets will be lazily loaded. This means that they won't occupy any memory until you use them. The following example shows memory usage before and after loading the nycflights13 package. You can see that memory usage doesn't change until you inspect the flights dataset stored inside the package.

```
pryr::mem_used()
#> 24.4 MB
library(nycflights13)
pryr::mem_used()
#> 24.7 MB

invisible(flights)
pryr::mem_used()
#> 59.9 MB
```

I recommend that you always include `LazyData: true` in your `DESCRIPTION`. `devtools::create()` does this for you.

Often, the data you include in `data/` is a cleaned up version of raw data you've gathered from elsewhere. I highly recommend taking the time to include the code used to do this in the source version of your package. This will make it easy for you to update or reproduce your version of the data. I suggest that you put this code in `data-raw/`. You don't need it in the bundled version of your package, so also add it to `.Rbuildignore`. Do all this in one step with:

```
devtools::use_data_raw()
```

You can see this approach in practice in some of my recent data packages. I've been creating these as packages because the data will rarely change, and because multiple packages can then use them for examples:

- babynames (`https://github.com/hadley/babynames`)

- fueleconomy (`https://github.com/hadley/fueleconomy`)
- nasaweather (`https://github.com/hadley/nasaweather`)
- nycflights13 (`https://github.com/hadley/nycflights13`)
- usdanutrients (`https://github.com/hadley/usdanutrients`)

9.1.1 Documenting datasets

Objects in `data/` are always effectively exported (they use a slightly different mechanism than `NAMESPACE`'s but the details are not important). This means that they must be documented. Documenting data is like documenting a function with a few minor differences. Instead of documenting the data directly, you document the name of the dataset. For example, the roxygen2 block used to document the diamonds data in ggplot2 looks something like this:

```
#' Prices of 50,000 round cut diamonds.
#'
#' A dataset containing the prices and other attributes of almost 54,000
#' diamonds.
#'
#' @format A data frame with 53940 rows and 10 variables:
#' \describe{
#'   \item{price}{price, in US dollars}
#'   \item{carat}{weight of the diamond, in carats}
#'   ...
#' }
#' @source \url{http://www.diamondse.info/}
"diamonds"
```

There are two additional tags that are important for documenting datasets:

- `@format` gives an overview of the dataset. For data frames, you should include a definition list that describes each variable. It's usually a good idea to describe variables' units here.

- `@source` provides details of where you got the data, often a `\url{}`.

Never `@export` a data set.

9.2 Internal data

Sometimes functions need pre-computed data tables. If you put these in `data/` they'll also be available to package users, which is not appropriate. Instead, you can save them in `R/sysdata.rda`. For example, two coloured related packages, munsell (`https://github.com/cwickham/munsell`) and dichromat (`http://cran.r-project.org/web/packages/dichromat/index.html`), use `R/sysdata.rda` to store large tables of colour data.

You can use `devtools::use_data()` to create this file with the argument `internal = TRUE`:

```r
x <- sample(1000)
devtools::use_data(x, mtcars, internal = TRUE)
```

Again, to make this data reproducible it's a good idea to include the code used to generate it. Put it in `data-raw/`.

Objects in `R/sysdata.rda` are not exported (they shouldn't be), so they don't need to be documented. They're only available inside your package.

9.3 Raw data

If you want to show examples of loading/parsing raw data, put the original files in `inst/extdata`. When the package is installed, all files (and folders) in `inst/` are moved up one level to the top-level directory (so they can't have names like `R/` or `DESCRIPTION`). To refer to files in `inst/extdata` (whether installed or not), use `system.file()`. For example, the testdat (`https://github.com/ropensci/testdat`) package uses `inst/extdata` to store a UTF-8 encoded csv file for use in examples:

```r
system.file("extdata", "2012.csv", package = "testdat")
#> [1] ""
```

Beware: if the file does not exist, `system.file()` does not return an error - it just returns the empty string:

```r
system.file("extdata", "2010.csv", package = "testdat")
#> [1] ""
```

9.4 Other data

- Data for tests: it's ok to put small files directly in your test directory. But remember unit tests are for testing correctness, not performance, so keep the size small.

- Data for vignettes. If you want to show how to work with an already loaded dataset, put that data in `data/`. If you want to show how to load raw data, put that data in `inst/extdata`.

9.5 CRAN notes

Generally, package data should be smaller than a megabyte - if it's larger you'll need to argue for an exemption. This is usually easier to do if the data is in its own package and won't be updated frequently. You should also make sure that the data has been optimally compressed:

1. Run `tools::checkRdaFiles()` to determine the best compression for each file.

2. Re-run `devtools::use_data()` with `compress` set to that optimal value. If you've lost the code for recreating the files, you can use `tools::resaveRdaFiles()` to re-save in place.

Chapter 10

Compiled code

R is high-level, expressive language. But that expressivity comes at a price: speed. That's why incorporating a low-level, compiled language like C or C++ can powerfully complement your R code. While C and C++ often require more lines of code (and more careful thought) to solve the same problem, they can be orders of magnitude faster than R.

Unfortunately, teaching you how to program in C or C++ is beyond the scope of the book. If you'd like to learn, I recommend starting with C++ and the Rcpp package. Rcpp makes it easy to connect C++ to R. I'd also recommend using RStudio because it has many tools that facilitate the entire process. Start by reading my "High performance functions with Rcpp" (`http://adv-r.had.co.nz/Rcpp.html`), a freely available book chapter from Advanced R (`http://amzn.com/1466586966?tag=devtools-20`): it gently introduces the language by translating examples of familiar R code into C++. Next, check out the Rcpp book (`http://www.rcpp.org/book`) and the other resources listed in learning more (`http://adv-r.had.co.nz/Rcpp.html#rcpp-more`).

10.1 C++

To set up your package with Rcpp, run:

```
devtools::use_rcpp()
```

This will:

- Create a `src/` directory to hold your `.cpp` files.
- Add `Rcpp` to the `LinkingTo` and `Imports` fields in the `DESCRIPTION`.

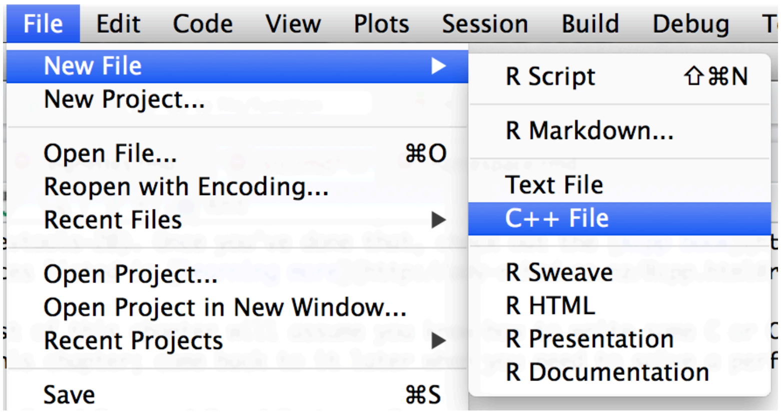
- Set up a `.gitignore` file to make sure you don't accidentally check in any compiled files (learn more about this in Chapter 13).
- Tell you the two roxygen tags you need to add to your package:

  ```
  #' @useDynLib your-package-name
  #' @importFrom Rcpp sourceCpp
  NULL
  #> NULL
  ```

10.1.1 Workflow

Once you're set up, the basic workflow should now be familiar:

1. Create a new C++ file:



   The default template looks like this:

   ```
   #include <Rcpp.h>
   using namespace Rcpp;

   // Below is a simple example of exporting a C++ function to R. You can
   // source this function into an R session using the Rcpp::sourceCpp
   // function (or via the Source button on the editor toolbar)

   // For more on using Rcpp click the Help button on the editor toolbar

   // [[Rcpp::export]]
   int timesTwo(int x) {
      return x * 2;
   }
   ```

It includes a basic function and some instructions to get started. The two most important parts are the header `#includes`, and the special attribute `// [[Rcpp::export]]`.

2. Generate the necessary modifications to your `NAMESPACE` by documenting them with Ctrl/Cmd + Shift + D.

3. Click Build & Reload in the build pane, or press Ctrl/Cmd + Shift + B. You can continue to use the standard `devtools::load_all()` process but it is more risky. Because you're loading and unloading C code, the chances of corrupting memory are high, and you're better off with the safer, but slower, "Build & Reload" which installs the package then restarts R.

4. Run `timesTwo(10)` from the console to check that it works.

Behind the scenes, "Build and reload" is doing a lot of work for you. They:

- Set up your R environment to compile code and warn you if you're missing necessary pieces.

- Call `Rcpp::compileAttributes()`. This inspects your `.cpp` functions looking for **attributes** of the form `// [[Rcpp::export]]`. When it finds one, it generates the code needed to make the function available in R, and creates `src/RcppExports.cpp` and `R/RcppExports.R`. You should never modify these files by hand.

- Build a DLL (dynamically linked library) and make it available to R.

10.1.2 Documentation

Each exported C++ function automatically gets a wrapper function (it will be located in `R/RcppExports.R`). For example, the R `timesTwo()` function looks like:

```
timesTwo <- function(x) {
  .Call('timesTwo', PACKAGE = 'mypackage', x)
}
```

This uses the base function `.Call()` to execute the C function `timesTwo` provided by "mypackage". You can use roxygen2 to document this like a regular R function. But instead of using `#'` for comments use `//'`, the C++ convention:

```
//' Multiply a number by two
//'
//' @param x A single integer.
//' @export
```

```
// [[Rcpp::export]]
int timesTwo(int x) {
   return x * 2;
}
```

That generates roxygen comments in `R/RcppExports.R`:

```
#' Multiply a number by two
#'
#' @param x A single integer.
#' @export
timesTwo <- function(x) {
  .Call('timesTwo', PACKAGE = 'mypackage', x)
}
```

The distinctions between the two export directives is important:

- `[[Rcpp::export]]` makes the C++ function available to R. If you have trouble remembering the exact details, note that everything comes in twos: Two `/`, two `[`, two `:` and two `]`.
- `@export` makes the R wrapper function available outside your package by adding it to the `NAMESPACE`.

10.1.3 Exporting C++ code

To make your C++ code callable from C++ code in other packages, add:

```
// [[Rcpp::interfaces(r, cpp)]]
```

This will generate a header file, `inst/include/mypackage.h` that can be included by other packages (The low-level details are described in Section 10.2.4{#c-export}). See "Rcpp Attributes (`http://dirk.eddelbuettel.com/code/rcpp/Rcpp-attributes.pdf`)" for more details, including how to combine hand-written and automatically generated header files.

10.1.4 Importing C++ code

To use C++ code from another package:

1. In `DESCRIPTION`, add `LinkingTo: otherPackage`. Confusingly this has nothing to do with the linker. It's called `LinkingTo` because it adds `otherPackage/include` to the include path, allowing you to dynamically "link to" other code via the headers.

2. In the C++ file, add:

   ```
   #include <otherPackage.h>
   ```

3. C++ functions from otherPackage will be included in the `otherPackage` namespace. Use `otherPackage::foo()` to access functions, or make them available globally with `using namespace otherPackage`.

10.1.5 Best practices

- To print output use `Rcout << ...` (not `cout << ...`). This prints to the right place, which might be a GUI console or a file (if `sink()` is active)
- In long-running loops, regularly run `Rcpp::checkUserInterrupt()`. This aborts your C++ if the user has pressed Ctrl + C or Escape in R.
- Use `.h` extension for headers and include files. (If you don't `R CMD check` will complain).
- Follow Martyn Plummer's recommendations on Portable C++ for R packages (`http://journal.r-project.org/archive/2011-2/RJournal_2011-2_Plummer.pdf`).
- Whenever you use C++ code in your package, you need to clean up after yourself when your package is unloaded. Do this by writing a `.onUnload()` function that unloads the DLL:

  ```
  .onUnload <- function (libpath) {
    library.dynam.unload("mypackage", libpath)
  }
  ```

- Use `clang` instead of `gcc` to compile your C++ code: it gives much better error messages. You can make `clang` the default by creating a `.R/Makevars` (linux and mac) or `.R/Makevars.win` (windows) file in your home directory that contains:

  ```
  CXX=clang++
  ```

  (If you don't know where your home directory is `path.expand("~")` will tell you.)

- To speed up compilation on linux or mac, install `ccache`, then replace `~/.R/Makevars` with:

  ```
  CC=ccache clang -Qunused-arguments
  CXX=ccache clang++ -Qunused-arguments
  CCACHE_CPP2=yes
  ```

10.2 C

If you're writing new compiled code, it's almost always better to use Rcpp. It's less work, more consistent, better documented, and it has better tools. However, there are some reasons to choose C:

- You're working with an older package that already uses the C API.
- You're binding to an existing C library.

There are two ways to call C functions from R: `.C()` and `.Call()`. `.C()` is a quick and dirty way to call an C function that doesn't know anything about R because `.C()` automatically converts between R vectors and the corresponding C types. `.Call()` is more flexible, but more work: your C function needs to use the R API to convert its inputs to standard C data types.

10.2.1 Getting started with `.Call()`

To call a C function from R, you first need a C function! In an R package, C code lives in `.c` files in `src/`. You'll need to include two header files:

```
#include <R.h>
#include <Rinternals.h>
```

(Yes, including `<Rinternals.h>` seems like bad form. On top of that, doing so doesn't actually give you access to the "internal" internal API unless you set some additional flags. The default just gives you access to the "public" internal API, which is both necessary and done for safety's sake. Yes, this is confusing.)

These headers allow you to access R's C API. Unfortunately this API is not well documented. I'd recommend starting with my notes at R's C interface (`http://adv-r.had.co.nz/C-interface.html`). After that, read "The R API (`http://cran.rstudio.com/doc/manuals/r-devel/R-exts.html#The-R-API`)" in "Writing R Extensions". A number of exported functions are not documented, so you'll also need to read the R source code (`https://github.com/wch/r-source`) to figure out the details.

Here's the bare minimum you need to know: C functions that talk to R must use the `SEXP` type for both inputs and outputs. `SEXP`, short for S expression, is the C struct used to represent every type of object in R. A C function typically starts by converting `SEXP`s to atomic C objects, and ends by converting C objects back to a `SEXP`. (The R API is designed so that these conversions often don't require copying.) The following table lists the functions that convert length one R vectors to and from C scalars:

R type	C type	R -> C	C -> R
integer	int	`asInteger(x)`	`ScalarInteger(x)`
numeric	double	`asReal(x)`	`ScalarReal(x)`
logical	int	`asLogical(x)`	`ScalarLogical(x)`
character	const char*	`CHAR(asChar(x))`	`mkString(x)`

We now have enough information to write a simple C function that can add two numbers:

```
#include <R.h>
#include <Rinternals.h>

SEXP add_(SEXP x_, SEXP y_) {
  double x = asReal(x_);
  double y = asReal(y_);

  double sum = x + y;

  return ScalarReal(sum);
}
```

We call this from R with `.Call()`:

```
#' @useDynLib mypackage add_
add <- function(x, y) .Call(add_, x, y)
```

Where does the first argument to `.Call()`, `add_`, come from? It comes from `@useDynLib`, which creates a line in the NAMESPACE that looks like:

```
useDynLib(mypackage, add_)
```

This directive instructs R to create an object called `add_` which describes a C function pointer:

```
mypackage:::add_
#> $name
#> [1] "add_"
#>
#> $address
#> <pointer: 0x107be3f40>
#> $package
#> NULL
#>
#> attr(,"class")
#> [1] "NativeSymbolInfo"
```

.Call() takes the pointer to a C function and calls it. All R objects have the same C type (the SEXP) you need to make sure the arguments are of the type you expect. Either do that in the R function, in the C function, or just accept that R will crash every time you accidentally supply the wrong type of input.

The most complicated part of working with the .Call() interface is memory-management. Whenever you create an R-level data structure, you must PROTECT() it so the garbage collector doesn't try and free it, then UNPROTECT() it at the end of the function. This topic is beyond the scope of this chapter, but you can learn more about it at http://adv-r.had.co.nz/C-interface.html#c-vectors.

10.2.2 Getting starting with .C()

.C() is simpler than .Call() and can be useful if you already have standard C code. Since you never create R objects in .C(), you never need to worry about memory management. To use it, you first write a void C function, using in-place modification of function parameters to return values:

```
void add_(double* x, double* y, double* out) {
  out[0] = x[0] + y[0];
}
```

Then like .Call() you create an R wrapper:

```
#' @useDynLib mypackage add_
add <- function(x, y) {
  .C(add_, x, y, numeric(1))[[3]]
}
```

(Here we extract the 3rd element of the result because that corresponds to the out parameter.)

.C() automatically converts back and forth between R vectors and their C equivalents:

R type	C type
logical	int*
integer	int*
double	double*
character	char**
raw	unsigned char*

Note that `.C()` assumes your function doesn't know how to deal with missing values and will throw an error if any arguments contain an NA. If it can correctly handle missing values, set `NAOK = TRUE` in the call to `.C()`.

You can learn more about `.C()` in its help, `?.C` and in R-extensions (`http://cran.rstudio.com/doc/manuals/r-devel/R-exts.html#Interface-functions-_002eC-and-_002eFortran`).

10.2.3 Workflow

The usual workflow still applies:

1. Modify the C code.
2. Build and reload the package with Ctrl/Cmd + Shift + B
3. Experiment at the console.

The first time you add `@useDynLib`, you'll also need to run `devtools::document()` (Ctrl/Cmd + Shift + D) and reload the package.

10.2.4 Exporting C code

R packages need to provide DLLs that can be relocated; DLLs that work regardless of where they live on disk. This is because most R users don't build packages from source. Instead, they get binaries from CRAN that can get installed in many different places. This need for relocatable DLLs adds a few more steps to the job of importing and exporting C code for R packages (the same problem arises for C++, but Rcpp attributes automate the manual steps described below).

R solves this problem using **function registration**. To export a `.Call()` C function, you register it with `R_RegisterCCallable()`. To import a `.Call()` C function, you get a pointer to it with `R_GetCCallable()`. Similar techniques are available for `.C()` C functions, but are beyond the scope of this book. As we'll see below, a user-friendly package will do both these tasks, so users of the package can ignore the details and simply include a header a file.

[Sidebar: Confusingly, there's another type of function registration. Instead of registering C functions using the namespace (i.e. `@useDynLib pkg fun`), you can register them with `R_registerRoutines()` and `@useDynLib mypackage, .registration = TRUE`. To learn the details read Registering native extensions (`http://cran.r-project.org/doc/manuals/R-exts.html#Registering-native-routines`) in "Writing R extensions".]

To register a function, call `R_RegisterCCallable()`, defined in `<R_ext/Rdynload.h>`. Function registration should be done in a function called `R_init_<mypackage>`.

This function is called automatically when the "mypackage" DLL is loaded. `R_RegisterCCallable()` has three arguments:

- A pointer to the DLL.
- The name of the function.
- A pointer to the function, cast as `DL_FUNC` (i.e. a **d**ynamically **l**oaded **func**tion).

The following code registers the `add()` function defined above:

```
#include "add.h"
#include <R_ext/Rdynload.h>

void R_init_mypackage(DllInfo *info) {
  R_RegisterCCallable(info, "add_",  (DL_FUNC) &add_)
}
```

It doesn't matter where this code lives, but it's usually put in a file called `src/mypackage-init.c`.

To access a registered function from another package, call `R_GetCCallable()`. It has two arguments, the function name and the package name. It returns a function pointer. The function pointer has no type information, so it should always be wrapped in a helper function that defines the inputs:

```
#include <R_ext/Rdynload.h>
#include <R.h>
#include <Rinternals.h>

SEXP add_(SEXP x, SEXP y) {
  static SEXP(fun*)(SEXP, SEXP) = NULL;
  if (fun == NULL)
    fun = (SEXP(*)(SEXP, SEXP)) R_GetCCallable("add", "mypackage");
  return fun(x, y);
}
```

Rather than relying on each package that imports your C code to do this correctly, you should instead do it for them. Write `inst/include/mypackageAPI.h` which provides a wrapper function for each exported function. A popular package that does that is xts (`http://cran.r-project.org/web/packages/xts/`). Download the source package and look in the `include/` directory to see what it does.

10.2.5 Importing C code

Using C code from another package varies based on how the package is implemented:

- If it uses the system described above, all you need is `LinkingTo: otherPackage` in the `DESCRIPTION`, and `#include otherPackageAPI.h` in the C file. (Remember `LinkingTo` is not about the linker, but actually affects the include path).
- If it registers the functions, but doesn't provide a header file, you'll need to write the wrapper yourself. Since you're not using any header files from the package, use `Imports` and not `LinkingTo`. You also need to make sure the package is loaded. You can do this by importing any function with `@importFrom mypackage foo`, or by adding `requireNamespace("mypackage", quietly = TRUE)` to `.onLoad()`.
- If it doesn't register the functions, you can't use them. You'll have to ask the maintainer nicely or even provide a pull request.

10.2.6 Best practices

- Avoid calls to `assert()`, `abort()` and `exit()`: these will kill the R process, not just your C code. Instead, use `error()` which is equivalent to calling `stop()` in R.
- To print output use `Rprintf()`, not `printf()`. Doing so always prints to the right place, whether it's the GUI console or a file (if `sink()` is active).
- In long-running loops, regularly call `R_CheckUserInterrupt()` to allow the user to interrupt the C code.
- Don't use C's random number generators (like `rand()` or `random()`), instead use the C API to R's rngs: `unif_rand()`, `norm_rand()`, etc. Note the caveats in "Random number generation" (`http://cran.rstudio.com/doc/manuals/r-devel/R-exts.html#Random-numbers`) - you must call `GetRNGstate()` before and `PutRNGstate()` after.
- Use R macros `ISNAN(x)` and `R_FINITE(x)` to check for NaNs and infinite values. These work on more platforms than the C99 `isnan()` and `isfinite()`.
- Like with C++, whenever you use C code in your package, you should unload the DLL when the package is unloaded:

  ```
  .onUnload <- function (libpath) {
    library.dynam.unload("mypackage", libpath)
  }
  ```

- Use clang instead of gcc to compile your C code: it gives much better error messages. You can make clang the default by creating a ~/.R/Makevars that contains:

  ```
  C=clang
  ```

10.3 Debugging compiled code

It's possible, with a little extra work, to use an interactive debugger to debug your C/C++ in the same way that you can use browser() and debug() to debug your R code. Unfortunately you won't be able to use RStudio, you'll have to run R from the command line.

Open a shell (e.g. with Tools | Shell…) and start R by typing:

```
# If you compile with clang
R --debugger=lldb
# If you compile with gcc
R --debugger=gdb
```

This will start either lldb (http://lldb.llvm.org) or gdb (http://www.gnu.org/software/gdb/), the debuggers that work with code produced by clang or gcc respectively. Like R, lldb and gdb provide a REPL, a run-eval-print loop where you enter commands and then look at the results. In the examples below I'll show the results of lldb, which is what I use (the output from gdb is similar). For each interactive command I'll tell you the explicit, but long, lldb command and the short, but cryptic, gdb command. Because lldb understand all gdb commands, you can use choose to be explicit of terse.

Once you've started the debugger, start R by typing process start (lldb) or run (gdb). Now when your C/C++ code crashes you'll be dumped into an interactive debugger instead of getting a cryptic error message and a crash.

Let's start with a simple C++ function that writes to memory it doesn't "own":

```
Rcpp::cppFunction("
bool mistake() {
  NumericVector x(1);
  int n = INT_MAX;
  x[n] = 0;
  return true;
}
", plugins = "debug", verbose = TRUE, rebuild = TRUE)
mistake()
```

Use `devtools::load_all()` to load the current package. Then copy and paste the code that creates the bug. Here's a crash report from a package I was working on:

```
Process 32743 stopped
* thread #1: tid = 0x1f79f6, 0x... gggeom.so...`
  frame #0: 0x0.. gggeom.so`vw_distance(x=..., y=...) + ... at vw-distance.cpp:54
   51        int prev_idx = prev[idx];
   52
   53       next[prev[idx]] = next_idx;
-> 54       prev[next[idx]] = prev_idx;
   55       prev[idx] = -1;
   56       next[idx] = -1;
   57
```

It tells us that the crash occurred because of a `EXC_BAD_ACCESS` - this is one of the most common types of crash in C/C++ code. Helpfully, lldb shows exactly which line of C++ code caused the problem: `vw-distance.cpp:54`. Often just knowing where the problem occurs is enough to fix it. But we're also now at an interactive prompt. There are many commands you can run here to explore what's going on. The most useful are listed below:

- See a list of all commands: `help`.
- Show your location on the callstack with `thread backtrace`/`bt`. This will print a list of calls leading up to the error, much like `traceback()` does in R. Navigate the callstack with `frame select <n>`/`frame <n>`, or `up` and `down`.
- Evaluate the next expression with `thread step-over`/`next`, or step into it with `thread step-in`/`step`. Continue executing the rest of the code with `thread step-out`/`finish`
- Show all variables defined in the current frame with `frame variable`/ `info locals`, or print the value of a single variable with `frame variable <var>`/`p <var>`.

Instead of waiting for a crash to occur you can also set breakpoints in your code. To do so, start the debugger, run R, then:

1. Press `Ctrl + C`
2. Type `breakpoint set --file foo.c --line 12`/`break foo.c:12`.
3. `process continue`/`c` to go back to the R console. Now run the C code you're interested in, and the debugger will stop when it gets to the specified line.

Finally, you can also use the debugger if your code is stuck in an infinite loop. Press `Ctrl + C` to break into the debugger and you'll see which line of code is causing the problem.

10.4 Makefiles

While makefiles are beyond the scope of this book, they are a useful tool. A good, gentle introduction with a focus on reproducible research is Karl Broman's "Minimal make" (`http://kbroman.org/minimal_make/`).

Generally, R packages should avoid a custom `Makefile`. Instead, use `Makevars`. `Makevars` is a make file that overrides the default make file generated by R (which is located at `file.path(R.home("etc"), "Makeconf")`). This allows you to take advantage of R's default behaviour (it's over 150 lines, and battle-tested across many years and many systems, so you want to!) while being able to set the flags you need. The most commonly used flags are:

- `PKG_LIBS`: Linker flags. A common use is `PKG_LIBS = $(BLAS_LIBS)`. This allows you to use the same BLAS library as R.
- `PKG_CFLAGS` & `PKG_CXXFLAGS`: C and C++ flags. Most commonly used to set define directives with `-D`.
- `PKG_CPPFLAGS`: Pre-processor flags (not C++ flags!). Most commonly used to set include directories with `-I`. Any package listed in the `LinkingTo` field in the `DESCRIPTION` will be automatically included - you do not need to explicitly add it.

To set flags only on Windows, use `Makevars.win`. To build a `Makevars` with `configure`, use `Makevars.in`.

By default, R will use the system make, which is not always GNU compatible (i.e. on Solaris). If you want to use GNU extensions (which are extremely common), add `SystemRequirements: GNU make` to `DESCRIPTION`. If you're not sure if you're using GNU extensions, play it safe and add it to the system requirement.

10.5 Other languages

It is possible to connect R to other languages, but the interfaces are not as nice as the one for C++:

- **Fortran**: It's possible to call Fortran subroutines directly with `.Fortran()`, or via C or C++ with `.Call()`. See `?.Fortran` and

- **Java**: The rJava (`https://github.com/s-u/rJava`) package makes it possible to call Java code from within R. Note that unlike with C and C++, passing an R object to a Java call will involve a copy operation, something which has serious performance implications.

10.6 Licensing

Because it's common to use other peoples' libraries when writing compiled code, you need to make sure that your package license is compatible with the licenses of all included code:

- The simplest solution is to use the same license as the included code. Since you can't relicense someone else's code, you may need to change your license.
- If you don't want to use the same license, you're best sticking with common cases where the interactions are well known. For example, Various Licenses and Comments about Them (`https://www.gnu.org/licenses/license-list.html`) describes what licenses are compatible with the GPL license.

In this case your description should contain `License: <main license> + FILE license` where `<main license>` is a license that is valid for the entire package (both R and compiled code), and the `license` file describes the licenses of individual components.

- For non-standard cases, you'll need to consult a lawyer.

In all cases, make sure you include copyright and license statements from the original code.

10.7 Development workflow

When developing C or C++ code, it's usually better to use RStudio's Build & Reload instead of `devtools::load_all()`. If you have C objects that persist between reloads, and you change the data structure, then it's

10.8 CRAN issues

Packages with compiled code are much more likely to have difficulties getting on CRAN than those without. The reason? Your package must build from source on all major platforms (Linux, Mac and Windows). This is hard!

- CRAN provides an automated service for checking R packages on windows: win-builder (`http://win-builder.r-project.org`). You can easily access this by running `devtools::build_win()`, which builds and uploads a package bundle.
- I've tried to include the most important advice in this chapter, but I'd recommend reading the entire section on writing portable C and C++ code (`http://cran.rstudio.com/doc/manuals/r-devel/R-exts.html#Portable-C-and-C_002b_002b-code`) in "Writing R extensions".
- In exceptional circumstances, like binding to Windows-only functionality, you may be able to opt-out of cross-platform requirement, but be prepared to make a strong case for it.

The interface between CRAN's automated and manual checking can be particularly frustrating for compiled code. Requirements vary from submission to submission, based on which maintainer you get and how much free time they have. The rules are inconsistently applied, but if your package doesn't pass, it's better to bite the bullet and make the change rather than trying to argue about it:

- Sometimes you will need to list all authors and copyright holders of included code in the DESCRIPTION.
- Sometimes your package will need to work on Solaris. But due to the difficulty of accessing a computer running Solaris, fixing Solaris issues can be hard. However, you will be in a stronger negotiating position if the package has no problems on other platforms.

  One common gotcha: gcc/clang flags `-Wall`, `-pedantic` and `-O0` do not work with the default compiler on Solaris.

Chapter 11

Installed files

When a package is installed, everything in `inst/` is copied into the top-level package directory. In some sense `inst/` is the opposite of `.Rbuildignore` - where `.Rbuildignore` lets you remove arbitrary files and directories to the top level, `inst/` lets you add them. You are free to put anything you like in `inst/` with one caution: because `inst/` is copied into the top-level directory, you should never use a subdirectory with the same name as an existing directory. This means that you should avoid `inst/build`, `inst/data`, `inst/demo`, `inst/exec`, `inst/help`, `inst/html`, `inst/inst`, `inst/libs`, `inst/Meta`, `inst/man`, `inst/po`, `inst/R`, `inst/src`, `inst/tests`, `inst/tools` and `inst/vignettes`.

This chapter discusses the most common files found in `inst/`:

- `inst/AUTHOR` and `inst/COPYRIGHT`. If the copyright and authorship of a package is particularly complex, you can use plain text files `inst/COPYRIGHTS` and `inst/AUTHORS` to provide more information.
- `inst/CITATION`: how to cite the package, see Section 11.1 for details.
- `inst/docs`: This is an older convention for vignettes, and should be avoided in modern packages.
- `inst/extdata`: additional external data for examples and vignettes. See Section 9.3 for more detail.
- `inst/java`, `inst/python` etc. See Section 11.2.

To find a file in `inst/` from code use `system.file()`. For example, to find `inst/extdata/mydata.csv`, you'd call `system.file("extdata", "mydata.csv", package = "mypackage")`. Note that you omit the `inst/` directory from the path. This will work if the package is installed, or if it's been loaded with `devtools::load_all()`.

11.1 Package 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:

```
citation()
#>
#> To cite R in publications use:
#>
#>   R Core Team (2015). R: A language and environment
#>   for statistical computing. R Foundation for
#>   Statistical Computing, Vienna, Austria. 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 = {2015},
#>     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.
```

Calling it with a package name tells you how to cite that package:

```
citation("lubridate")
#>
#> To cite lubridate in publications use:
#>
#>   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/.
#>
#> A BibTeX entry for LaTeX users is
#>
#>   @Article{,
#>     title = {Dates and Times Made Easy with {lubridate}},
#>     author = {Garrett Grolemund and Hadley Wickham},
#>     journal = {Journal of Statistical Software},
#>     year = {2011},
#>     volume = {40},
#>     number = {3},
#>     pages = {1--25},
#>     url = {http://www.jstatsoft.org/v40/i03/},
#>   }
```

To customise the citation for your package, add a `inst/CITATION` that looks like this:

```
citHeader("To cite lubridate in publications use:")

citEntry(entry = "Article",
  title        = "Dates and Times Made Easy with {lubridate}",
  author       = personList(as.person("Garrett Grolemund"),
                   as.person("Hadley Wickham")),
  journal      = "Journal of Statistical Software",
  year         = "2011",
  volume       = "40",
  number       = "3",
  pages        = "1--25",
  url          = "http://www.jstatsoft.org/v40/i03/",

  textVersion  =
  paste("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/.")
)
```

You need to create `inst/CITATION`. As you can see, it's pretty simple: you only need to learn one new function, `citEntry()`. The most important arguments are:

- `entry`: the type of citation, "Article", "Book", "PhDThesis" etc.
- The standard bibliographic information like `title`, `author` (which should be a `personList()`), `year`, `journal`, `volume`, `issue`, `pages`, …

A complete list of arguments can be found in `?bibentry`.

Use `citHeader()` and `citFooter()` to add additional exhortations.

11.2 Other languages

Sometimes a package contains useful supplementary scripts in other programming languages. Generally, you should avoid these, because it adds an additional extra dependency, but it may be useful when wrapping substantial amounts of code from another language. For example, gdata (`http://cran.r-project.org/web/packages/gdata/index.html`) wraps the Perl module Spreadsheet::ParseExcel (`http://search.cpan.org/~dougw/Spreadsheet-ParseExcel-0.65/`) to read excel files into R.

The convention is to put scripts of this nature into a subdirectory of `inst/`, `inst/python`, `inst/perl`, `inst/ruby` etc. If these scripts are essential to your package, make sure you also add the appropriate programming language to the `SystemRequirements` field in the `DESCRIPTION`. (This field is for human reading so don't worry about exactly how you specify it.)

Java is a special case because you need to include both the source code (which should go in `java/` and be listed in `.Rinstignore`), and the compiled jar files (which should go in `inst/java`). Make sure to add `rJava` to `Imports`.

Chapter 12

Other components

There are four other directories that are valid top-level directories. They are rarely used:

- `demo/`: for package demos. These were useful prior to the introduction of vignettes, but are no longer recommended. See below.
- `exec/`: for executable scripts. Unlike files placed in other directories, files in `exec/` are automatically flagged as executable.
- `po/`: translations for messages. This is useful but beyond the scope of this book. See the Internationalization (`http://cran.rstudio.com/doc/manuals/r-devel/R-exts.html#Internationalization`) chapter of "R extensions" for more details.
- `tools/`: auxiliary files needed during configuration, or for sources that need to generate scripts.

12.1 Demos

A demo is an `.R` file that lives in `demo/`. Demos are like examples but tend to be longer. Instead of focussing on a single function, they show how to weave together multiple functions to solve a problem.

You list and access demos with `demo()`:

- Show all available demos: `demo()`.
- Show all demos in a package: `demo(package = "httr")`.
- Run a specific demo: `demo("oauth1-twitter", package = "httr")`.

- Find a demo: `system.file("demo", "oauth1-twitter.R", package = "httr")`.

Each demo must be listed in `demo/00Index` in the following form: `demo-name Demo description`. The demo name is the name of the file without the extension, e.g. `demo/my-demo.R` becomes `my-demo`.

By default the demo asks for human input for each plot: "Hit to see next plot:". This behaviour can be overridden by adding `devAskNewPage(ask = FALSE)` to the demo file. You can add pauses by adding: `readline("press any key to continue")`.

Generally, I do not recommend using demos. Instead, consider writing a vignette:

- Demos are not automatically tested by `R CMD check`. This means that they can easily break without your knowledge.
- Vignettes have both input and output, so readers can see the results without having to run the code themselves.
- Longer demos need to mingle code with explanation, and RMarkdown is better suited to that task than R comments.
- Vignettes are listed on the CRAN package page. This makes it easier for new users to discover them.

Part III

Best practices

Chapter 13

Git and GitHub

If you're serious about software development, you need to learn about Git. Git is a *version control system*, a tool that tracks changes to your code and shares those changes with others. Git is most useful when combined with GitHub (`http://github.com`), a website that allows you to share your code with the world, solicit improvements via pull requests and track issues. Git + GitHub is the most popular version control system for developers of R packages (witness the thousands of R packages hosted on GitHub).

Git and GitHub are generally useful for all software development and data analysis, not just R packages. I've included it here, because it is so useful when you're making a package. There's no way I could be as productive without Git and GitHub at my back, enabling me to rapidly spot mistakes and easily collaborate with others.

Why use Git + GitHub?

- It makes sharing your package easy. Any R user can install your package with just two lines of code:

  ```
  install.packages("devtools")
  devtools::install_github("username/packagename")
  ```

- GitHub is a great way to make a barebones website for your package. Readers can easily browse code, and read documentation (via Markdown). They can report bugs, suggest new features with GitHub issues (`https://guides.github.com/features/issues/`), and propose improvements to your code with pull requests.

- Have your ever tried to collaboratively write code with someone by sending files back and forth via email or a Dropbox folder? It takes a lot of effort just to make sure that the two of you aren't working on the same file and overwriting each others' changes. With Git, both of you can work on the same file at the same time. Git will either combine your changes automatically, or it will show you all the ambiguities and conflicts.

- Have you ever accidentally pressed `s` instead of Cmd + S to save your file? It's very easy to accidentally introduce a mistake that takes a few minutes to track down. Git makes this problem easy to spot because it allows you to see exactly what's changed and undo any mistakes.

You can do many of these same things with other tools (like subversion (`https://subversion.apache.org`) or mercurial (`http://mercurial.selenic.com`)) and other websites (like gitlab (`https://about.gitlab.com`) and bitbucket (`https://bitbucket.org`)). Git is most useful in conjunction with GitHub, and vice versa, so I'll make no effort to distinguish between features that belong to Git and those that belong to GitHub. But I think Git + GitHub is the most user-friendly system (especially for new developers), not least because its popularity means that the answer or solution to every possible question or problem can be found on StackOverflow.

This is not to say that Git is easy to learn. Your initial experiences with Git are likely to be frustrating and you will frequently curse at the strange terminology and unhelpful error messages. Fortunately, there are many tutorials available online, and while they aren't always well written (many provide a lot of information but little guidance about what to do with it or why you need to care), you can absolutely master Git with a little practice. Don't give up! Persevere and you'll unlock the super-power of code collaboration.

13.1 RStudio, Git and GitHub

RStudio makes day-to-day use of Git simpler. Once you've set up a project to use Git, you'll see a new pane and toolbar icon. These provide shortcuts to the most commonly used Git commands. However, because only a handful of the 150+ Git commands are available in RStudio, you also need to be familiar with using Git from the **shell** (aka the command line or the console). It's also useful to be familiar with using Git in a shell because if you get stuck you'll need to search for a solution with the Git command names.

The easiest way to get to a shell from RStudio is Tools > Shell. This will open a new shell located in the root directory of your project. (NB: on Windows, this opens up a *bash* shell, the standard Linux shell, which behaves a little differently from the usual `cmd.exe` shell.)

Don't worry if you've never used the shell before because it's very similar to using R. The main difference is that instead of functions, you call commands, which have a slightly different syntax. For example, in R you might write `f(x, y = 1)`, where in the shell you'd write `f x --y=1` or `f x -y1`. Also, while shell commands are even less regular than R functions, you fortunately only need to be familiar with a few. In this chapter, you won't be doing much in the shell apart from running Git commands. However, it's a good idea to learn the three most important shell commands:

- `pwd`: print working directory. This tells you which directory you're currently in.
- `cd <name>`: change directory. Use `cd ..` to move up the directory hierarchy.
- `ls`: list files. Shows all files in the current directory.

If you've never used the shell before, I recommend playing Terminus (`http://web.mit.edu/mprat/Public/web/Terminus/Web/main.html`). It's a fun way to learn the basics of the shell. I also recommend taking a look at Philip Guo's Basic Unix-like command line tutorial (`http://pgbovine.net/command-line-tutorial.htm`) videos, and at `http://www.ee.surrey.ac.uk/Teaching/Unix/unix1.html` and `https://p1k3.com/userland-book/`.

13.2 Initial set up

If you've never used Git or GitHub before, start by installing Git and creating a GitHub account. Then, link the two together:

1. Install Git:
   - Windows: `http://git-scm.com/download/win`.
   - OS X: `http://git-scm.com/download/mac`.
   - Debian/Ubuntu: `sudo apt-get install git-core`.
   - Other Linux distros: `http://git-scm.com/download/linux`.
2. Tell Git your name and email address. These are used to label each commit so that when you start collaborating with others, it's clear who made each change. In the shell, run:

   ```
   git config --global user.name "YOUR FULL NAME"
   git config --global user.email "YOUR EMAIL ADDRESS"
   ```

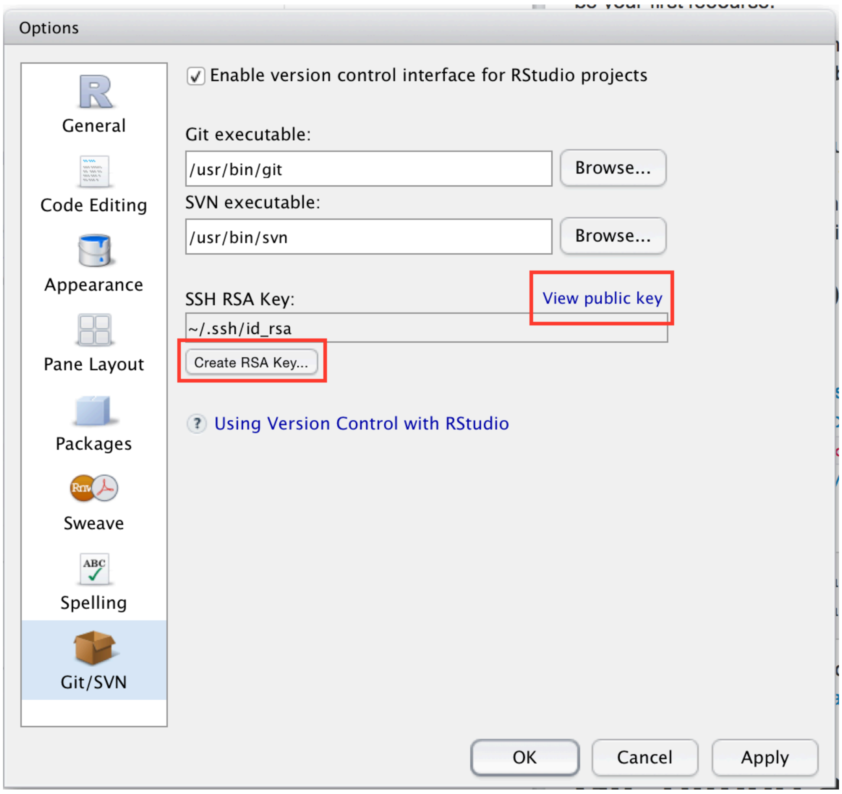
   (You can check if you're set up correctly by running `git config --global --list`.)
3. Create an account on GitHub, `https://github.com` (the free plan is fine). Use the same email address as above.

4. If needed, generate a SSH key. SSH keys allow you to securely communicate with websites without a password. There are two parts to an SSH key: one public, one private. People with your public key can securely encrypt data that can only be read by someone with your private key.

   You can check if you already have an SSH key-pair by running:

   ```
   file.exists("~/.ssh/id_rsa.pub")
   ```

   If that returns `FALSE`, you'll need to create a new key. You can either follow the instructions on GitHub (`https://help.github.com/articles/generating-ssh-keys`) or use RStudio. Go to RStudio preferences, choose the Git/SVN panel, and click "Create RSA key…":



☑ Enable version control interface for RStudio projects

Git executable: /usr/bin/git

SVN executable: /usr/bin/svn

SSH RSA Key: ~/.ssh/id_rsa

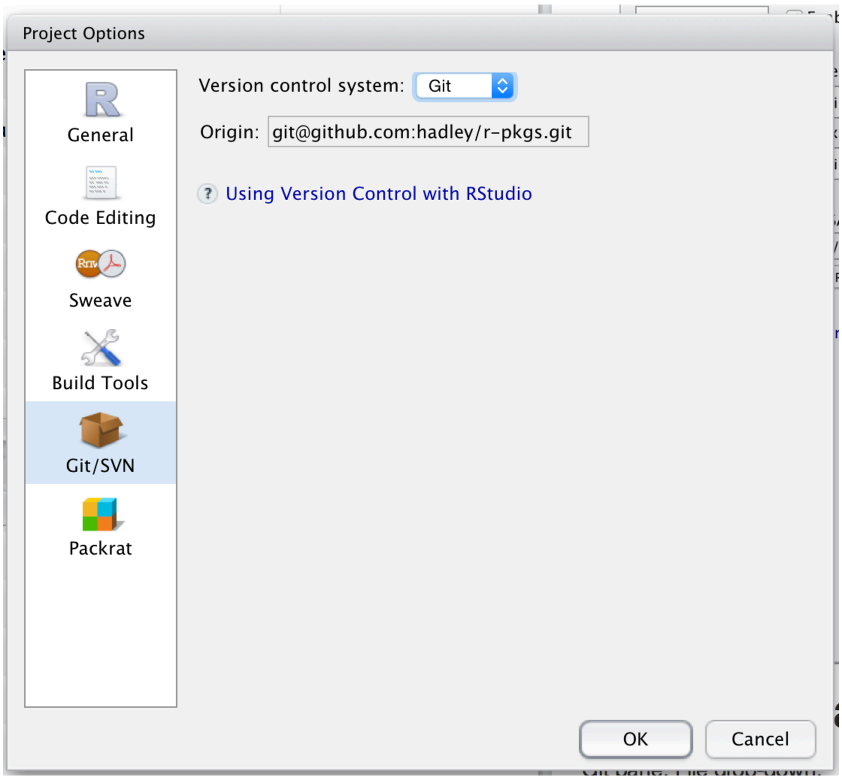
5. Give GitHub your SSH public key: `https://github.com/settings/ssh`. The easiest way to find the key is to click "View public key" in RStudio's Git/SVN preferences pane.

13.3 Create a local Git repository

Now that you have installed and configured Git, you can use it! To use GitHub with your package, you'll need to initialise a local repository, or **repo** for short. This creates a `.git` directory that stores configuration files and a database that records changes to your code. A new repo exists only on your computer; you'll learn how to share it with others shortly.

To create a new repo:

- In RStudio, go to project options, then to the Git/SVN panel. Change the "Version control system" from "None" to "Git":

  Version control system: Git

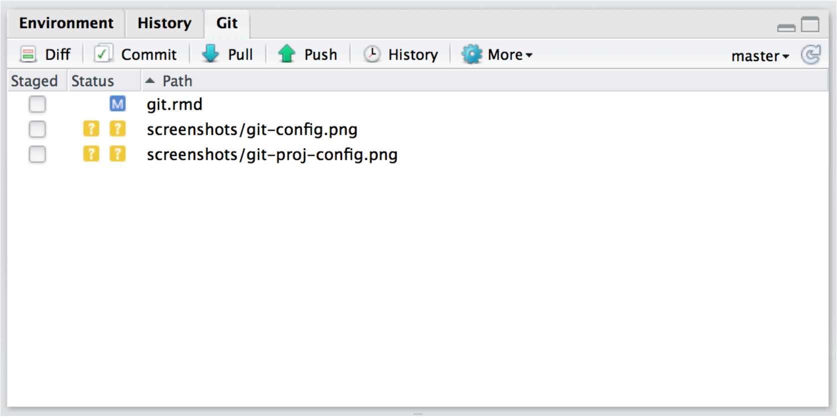
  Origin: git@github.com:hadley/r-pkgs.git

  You'll then be prompted to restart RStudio.

- In a shell, run `git init`. Restart RStudio and reopen your package.

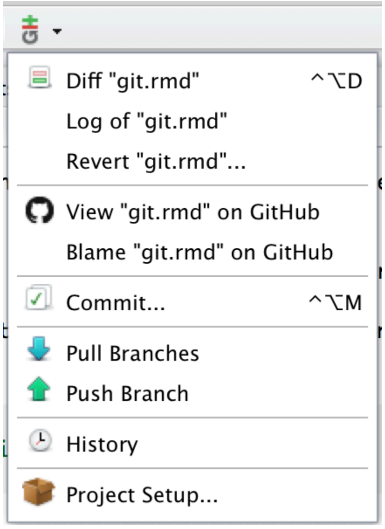
Once Git has been initialised, you'll see two new components:

- The **git pane**, at the top-right, shows you what files have changed and includes buttons for the most important Git commands:



Staged	Status	Path
□	M	git.rmd
□	? ?	screenshots/git-config.png
□	? ?	screenshots/git-proj-config.png

- The **git dropdown** menu, found in the toolbar, includes Git and GitHub commands that apply to the current file:



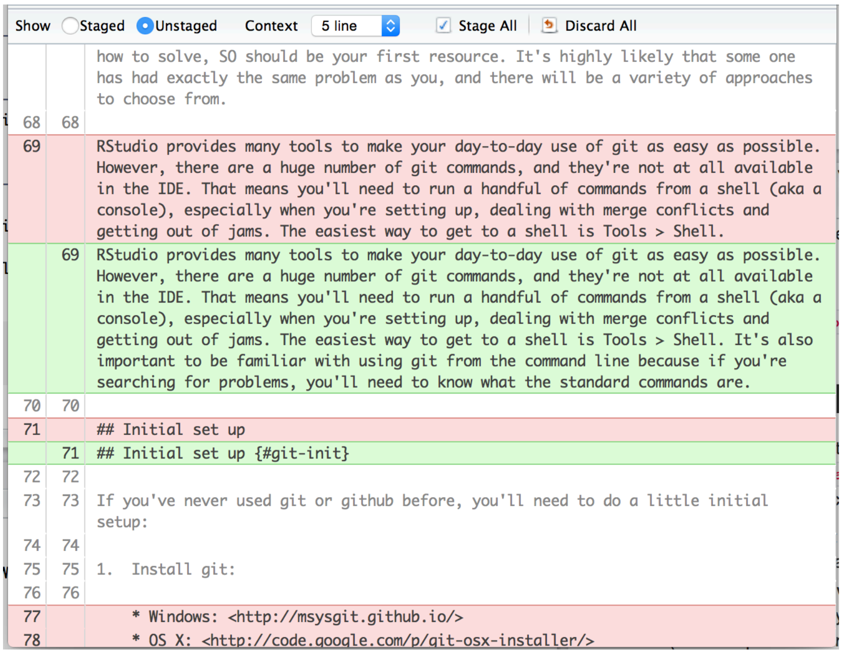
13.4 See what's changed

The first benefit of Git is that you can easily see the changes you've made. I find this really helpful, as I often accidentally mistype keyboard shortcuts, leaving stray characters in my code. The RStudio Git pane lists every file that's been added, modified or deleted. The icon describes the change:

- M, **Modified**. You've changed the contents of the file.
- ?, **Untracked**. You've added a new file that Git hasn't seen before.

- D, **Deleted**. You've deleted a file.

You can get more details about modifications with a "diff", Diff. This opens a new window showing the detailed **differences**:



Show ○ Staged ● Unstaged   Context 5 line   ☑ Stage All   Discard All

```
        how to solve, SO should be your first resource. It's highly likely that some one
        has had exactly the same problem as you, and there will be a variety of approaches
        to choose from.
68  68
69      RStudio provides many tools to make your day-to-day use of git as easy as possible.
        However, there are a huge number of git commands, and they're not at all available
        in the IDE. That means you'll need to run a handful of commands from a shell (aka a
        console), especially when you're setting up, dealing with merge conflicts and
        getting out of jams. The easiest way to get to a shell is Tools > Shell.
    69  RStudio provides many tools to make your day-to-day use of git as easy as possible.
        However, there are a huge number of git commands, and they're not at all available
        in the IDE. That means you'll need to run a handful of commands from a shell (aka a
        console), especially when you're setting up, dealing with merge conflicts and
        getting out of jams. The easiest way to get to a shell is Tools > Shell. It's also
        important to be familiar with using git from the command line because if you're
        searching for problems, you'll need to know what the standard commands are.
70  70
71      ## Initial set up
    71  ## Initial set up {#git-init}
72  72
73  73  If you've never used git or github before, you'll need to do a little initial
        setup:
74  74
75  75  1.  Install git:
76  76
77          * Windows: <http://msysgit.github.io/>
78          * OS X: <http://code.google.com/p/git-osx-installer/>
```

The background colours tells you whether the text has been added (green) or removed (red). (If you're colourblind you can use the line numbers in the two columns at the far left as a guide: a number in the first column identifies the old version, a number in second column identifies the new version.) The grey lines of code above and below the changes give you additional context.

In the shell, use `git status` to see an overview of changes and `git diff` to show detailed differences.

13.5 Record changes

The fundamental unit of work in Git is a **commit**. A commit takes a snapshot of your code at a specified point in time. Using a Git commit is like using anchors and other protection when climbing. If you're crossing a dangerous rock face you want to make sure you've used protection to catch you if you fall. Commits play a similar role: if you make a mistake, you can't fall past the previous commit. Coding without commits is like free-climbing: you can travel much faster in the short-term, but in the long-term the chances of catastrophic failure are high! Like rock climbing protection, you want to be judicious in your use of commits. Committing too frequently will slow your progress; use more commits when you're in uncertain or dangerous territory. Commits are also helpful to others, because they show your journey, not just the destination.

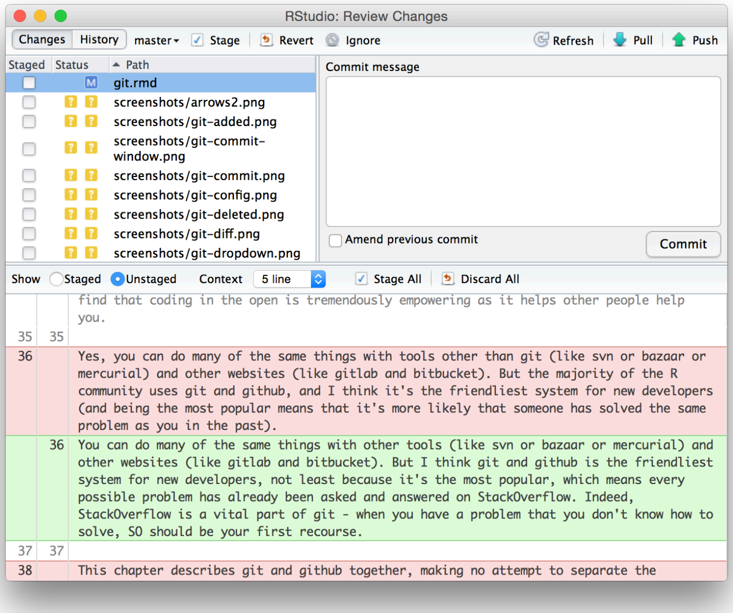
There are five key components to every commit:

- A unique identifier, called a SHA (short for secure hash algorithm).
- A changeset that describes which files were added, modified and deleted.
- A human-readable commit message.
- A parent, the commit that came before this one. (There are two exceptions to this rule: the initial commit doesn't have a parent, and merges, which you'll learn about later, have two parents.)
- An author.

You create a commit in two stages:

1. You **stage** files, telling Git which changes should be included in the next commit.
2. You **commit** the staged files, describing the changes with a message.

In RStudio, staging and committing are done in the same place, the commit window, which you can open by clicking ☑ Commit or by pressing Ctrl + Alt + m.



Staged	Status	Path
□	M	git.rmd
□	? ?	screenshots/arrows2.png
□	? ?	screenshots/git-added.png
□	? ?	screenshots/git-commit-window.png
□	? ?	screenshots/git-commit.png
□	? ?	screenshots/git-config.png
□	? ?	screenshots/git-deleted.png
□	? ?	screenshots/git-diff.png
□	? ?	screenshots/git-dropdown.png

```
find that coding in the open is tremendously empowering as it helps other people help
you.

Yes, you can do many of the same things with tools other than git (like svn or bazaar or
mercurial) and other websites (like gitlab and bitbucket). But the majority of the R
community uses git and github, and I think it's the friendliest system for new developers
(and being the most popular means that it's more likely that someone has solved the same
problem as you in the past).
You can do many of the same things with other tools (like svn or bazaar or mercurial) and
other websites (like gitlab and bitbucket). But I think git and github is the friendliest
system for new developers, not least because it's the most popular, which means every
possible problem has already been asked and answered on StackOverflow. Indeed,
StackOverflow is a vital part of git - when you have a problem that you don't know how to
solve, SO should be your first recourse.

This chapter describes git and github together, making no attempt to separate the
```

The commit window is made up of three panes:

- The top-left pane shows the current status in the way as the Git pane in the main RStudio window.
- The bottom pane shows the diff of the currently selected file.
- The top-right pane is where you'll enter the commit message, a human readable message summarising the changes made in the commit. More on that shortly.

(Yes, this is exactly the same window you see when clicking Diff!)

To create a new commit:

1. **Save your changes**.
2. **Open the commit window** by clicking ☑ Commit or pressing `Ctrl + Alt + m`.
3. **Select files**. To stage (select) a single file for inclusion, tick its check box.

   To stage all files, press Ctrl/Cmd + A, then click ☑ Stage.

   As you stage each file, you'll notice that its status changes. The icon will change columns from right (unstaged status) to left (staged status), and you might see one of two new icons:

   - Added: A: after staging an untracked file, Git now knows that you want to add it to the repo.

   - Renamed: R: If you rename a file, Git initially sees it as a deletion and addition. Once you stage both changes, Git will recognise that it's a rename.

   Sometimes you'll see a status in both columns, e.g. M M. This means that you have both staged and unstaged changes in the same file. This happens when you've made some changes, staged them, and then made some more. Clicking the staged checkbox will stage your new changes, clicking it again will unstage both sets of changes.

4. **Stage files**, as above.

5. **Write a commit message** (top-right panel) which describes the changes that you've made. The first line of a commit message is called the subject line and should be brief (50 characters or less). For complicated commits, you can follow it with a blank line and then a paragraph or bulleted list providing more detail. Write messages in imperative, like you're telling someone what to do: "fix this bug", not "fixed this bug" or "this bug was fixed".

6. **Click Commit**.

Staging files is a little more complicated in the shell. You use `git add` to stage new and modified files, and `git rm` to stage deleted files. To create the commit, use `git commit -m <message>`.

13.6 Commit best practices

Ideally, each commit should be *minimal* but *complete*:

- **Minimal**: A commit should only contain changes related to a single problem. This will make it easier to understand the commit at a glance, and to describe it with a simple message. If you should discover a new problem, you should do a separate commit.

- **Complete**: A commit should solve the problem that it claims to solve. If you think you've fixed a bug, the commit should contain a unit test that confirms you're right.

Each commit message should:

- **Be concise, yet evocative**. At a glance, you should be able to see what a commit does. But there should be enough detail so you can remember (and understand) what was done.
- **Describe the why, not the what**. Since you can always retrieve the diff associated with a commit, the message doesn't need to say exactly what changed. Instead it should provide a high-level summary that focuses on the reasons for the change.

If you do this:

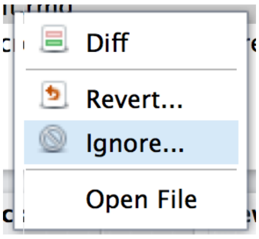
- It'll be easier to work with others. For example, if two people have changed the same file in the same place, it'll be easier to resolve conflicts if the commits are small and it's clear why each change was made.
- Project newcomers can more easily understand the history by reading the commit logs.
- You can load and run your package at any point along its development history. This can be tremendously useful with tools like bisectr (`https://github.com/wch/bisectr`), which allow you to use binary search to quickly find the commit that introduced a bug.
- If you can figure out exactly when a bug was introduced, you can easily understand what you were doing (and why!).

You might think that because no one else will ever look at your repo, that writing good commit messages is not worth the effort. But keep in mind that you have one very important collaborator: future-you! If you spend a little time now polishing your commit messages, future-you will thank you if and when they need to do a post-mortem on a bug.

Remember that these directives are aspirational. You shouldn't let them get in your way. If you look at the commit history of my repositories, you'll notice a lot of them aren't that good, especially when I start to get frustrated that I **still** haven't managed to fix a bug. Strive to follow these guidelines, and remember it's better to have multiple bad commits than to have one perfect commit.

13.7 Ignoring files

Often, there are files that you don't want to include in the repository. They might be transient (like LaTex or C build artefacts), very large, or generated on demand. Rather than carefully *not* staging them each time, you should instead add them to `.gitignore`. This will prevent them from ever being added. The easiest way to do this is to right-click on the file in the Git pane and select `Ignore`:



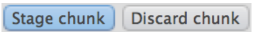
If you want to ignore multiple files, you can use a wildcard "glob" like `*.png`. To learn more about the options, see ignoring files (`http://git-scm.com/book/ch2-2.html#Ignoring-Files`) in Pro-Git.

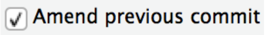
Some developers never commit derived files, files that can be generated automatically. For an R package this would mean ignoring the files in the `NAMESPACE` and `man/` directories because they're generated from comments. From a practical pespective, it's better to commit these files: R packages have no way to generate `.Rd` files on installation so ignoring derived files means that users who install your package from GitHub will have no documentation.

13.8 Undo a mistake

The best thing about using commits is that you can undo mistakes. RStudio makes it particularly easy:

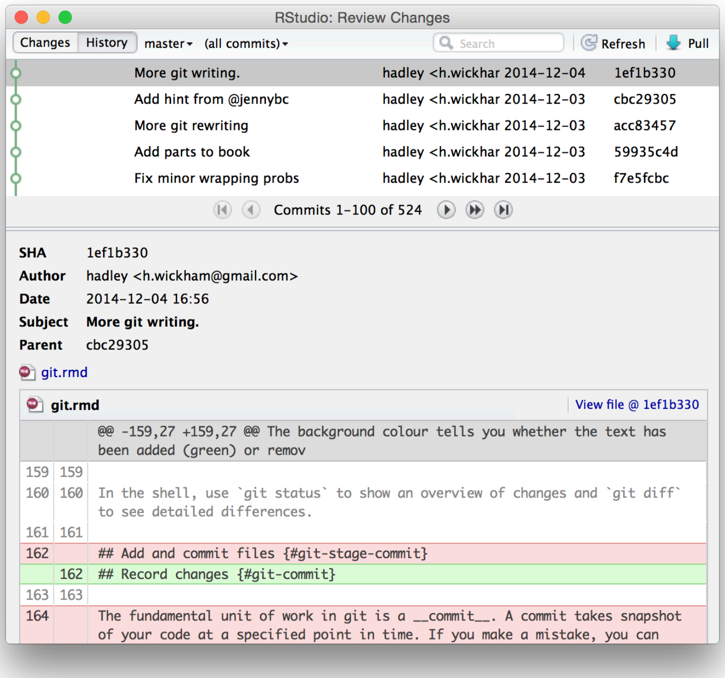
- To undo the changes you've just made, right click on the file in the Git pane and select "revert". This will roll any changes back to the previous commit. Beware: you can't undo this operation!

  You can also undo changes to just part of a file in the diff window. Look for a **discard chunk** button above the block of changes that you want to undo: . You can also discard changes to individual lines or selected text.

- If you committed changes too early, you can modify the previous commit by staging the extra changes. Before you click commit, select . (Don't do this if you've pushed the previous commit to GitHub - you're effectively rewriting history, which should be done with care when you're doing it in public.)

If you didn't catch the mistake right away, you'll need to look backwards in history and find out where it occurred:

1. Open the history window by clicking History in the Git pane.



More git writing.	hadley <h.wickhar 2014-12-04	1ef1b330
Add hint from @jennybc	hadley <h.wickhar 2014-12-03	cbc29305
More git rewriting	hadley <h.wickhar 2014-12-03	acc83457
Add parts to book	hadley <h.wickhar 2014-12-03	59935c4d
Fix minor wrapping probs	hadley <h.wickhar 2014-12-03	f7e5fcbc

SHA 1ef1b330
Author hadley <h.wickham@gmail.com>
Date 2014-12-04 16:56
Subject **More git writing.**
Parent cbc29305

git.rmd

git.rmd View file @ 1ef1b330

```
        @@ -159,27 +159,27 @@ The background colour tells you whether the text has
        been added (green) or remov
159 159
160 160 In the shell, use `git status` to show an overview of changes and `git diff`
        to see detailed differences.
161 161
162     ## Add and commit files {#git-stage-commit}
    162 ## Record changes {#git-commit}
163 163
164     The fundamental unit of work in git is a __commit__. A commit takes snapshot
        of your code at a specified point in time. If you make a mistake, you can
```

The history window is divided into two parts. The top part lists every commit to your repo. The bottom part shows you the commit: the SHA (the unique id), the author, the date, the parent and the changes in the commit.

2. Navigate back in time until you find the commit where the mistake occurred. Write down the parent SHA: that's the commit that occurred before the mistake so it will be good.

Now you can use that SHA in the shell:

- See what the file looked like in the past so you can copy-and-paste the old code:

  ```
  git show <SHA> <filename>
  ```

- Or copy the version from the past back in to the present:

  ```
  git checkout <SHA> <filename>
  ```

In both cases you'll need to finish by staging and committing the files.

(It's also possible to use Git as if you went back in time and prevented the mistake from happening in the first place. This is an advanced technique called **rebasing history**. As you might imagine, going back in time to change the past can have a profound impact on the present. It can be useful, but it needs to be done with extreme care.)

If you're still stuck, try `http://sethrobertson.github.io/GitFixUm/fixup.html` or `http://justinhileman.info/article/git-pretty/`. They give step-by-step approaches to fixing many common (and not so common!) problems.

13.9 Synchronising with GitHub

So far we've only been working locally, using commits to track the progress of a project and to provide safe checkpoints. However, Git really shines when you start sharing your code with others with GitHub (`http://github.com`). While there are other choices, I recommend GitHub because it is free for open source projects, it has all the features you'll need, and is a popular choice in the R world.

To publish, or **push**, your code to GitHub:

1. Create a new repo on GitHub: `https://github.com/new`. Give it the same name as your package, and include the package title as the repo description. Leave all the other options as is, then click Submit.

2. Open a shell, then follow the instructions on the new repo page. They'll look something like this:

   ```
   git remote add origin git@github.com:hadley/r-pkgs.git
   git push -u origin master
   ```

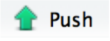
   The first line tells Git that your local repo has a remote version on GitHub, and calls it "origin". The second line pushes all your current work to that repo.

Now let's make a commit and verify that the remote repo updates:

1. Modify `DESCRIPTION` to add `URL` and `BugReports` fields that link to your new GitHub site. For example, dplyr has:

   ```
   URL: http://github.com/hadley/dplyr
   BugReports: http://github.com/hadley/dplyr/issues
   ```

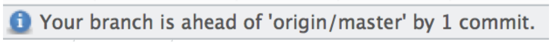
2. Save the file and commit (with the message "Updating DESCRIPTION to add links to GitHub site").

3. **Push** your changes to GitHub by clicking  Push . (This is the same as running `git push` in the shell).

4. Go to your GitHub page and look at the `DESCRIPTION`.

Usually, each push will include multiple commits. This is because you push much less often than you commit. How often you push versus commit is completely up to you, but pushing code means publishing code. So strive to push code that works.

To ensure your code is clean, I recommend always running `R CMD check` before you push (a topic you'll learn about in the chapter on Chapter 14). If you want to publish code that doesn't work (yet), I recommend using a branch, as you'll learn about below in Section 13.13.

Once you've connected your repo to GitHub, the Git pane will show you how many commits you have locally that are not on GitHub:  Your branch is ahead of 'origin/master' by 1 commit. . This message indicates that I have 1 commit locally (my branch) that is not on GitHub ("origin/master").

13.10 Benefits of using GitHub

1. You get a decent website. The GitHub page for your project, e.g. `https://github.com/hadley/testthat` (the GitHub repo for testthat), lists all the files and directories in your package. `.R` files will be formatted with syntax highlighting, and `.md`/`.Rmd` files will be rendered as HTML. And, if you include a `README.md` file in the top-level directory, it will be displayed on the homepage. You'll learn more about the benefits of creating this file in Section 15.5.1.

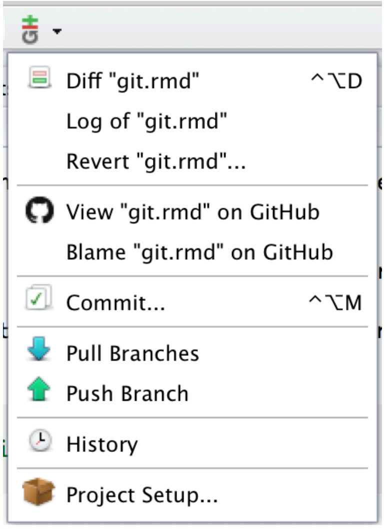
2. It makes it easy for anyone to install your package (and to benefit from your hard work):

   ```
   devtools::install_github("<your_username>/<your_package>")
   ```

3. You can track the history of the project in the commit view, e.g. `https://github.com/hadley/testthat/commits/master`. When I'm working on a package with others, I often keep this page open so I can see what they're working on. Individual commits show the same information that you see in the commit/diff window in RStudio.

4. It's easy to see the history of a file. If you navigate to a file and click **History**, you'll see every commit that affected that file. Another useful view is **Blame**; it shows the last change made to each line of code, who made the change, and the commit the change belongs to. This is tremendously helpful when you're tracking down a bug.

You can jump directly to these pages from RStudio with the Git dropdown in the main toolbar:



5. You can comment on commits. To comment on the commit as a whole, use the comment box at the bottom of the page. To comment on an individual line, click the plus sign that appears when you mouse over a line number, 3 ➕. This is a great way to let your collaborators know if you see a mistake or have a question. It's better than email because it's public so anyone working on the repo (both present and future) can see the conversation.

13.11 Working with others

You use **push** to send your changes to GitHub. If you're working with others, they also push their changes to GitHub. But, to see their changes locally you'll need to **pull** their changes from GitHub. In fact, to make sure everyone is in sync, Git will only let you push to a repo if you've retrieved the most recent version with a pull.

When you pull, Git first downloads (**fetches**) all of the changes and then **merges** them with the changes that you've made. A merge is a commit with two parents. It takes two different lines of development and combines them into a single result. In many cases, Git can do this automatically: for example, when changes are made to different files, or to different parts of the same file. However, if changes are made to the same place in a file, you'll need to resolve the **merge conflict** yourself.

In RStudio, you'll discover that you have merge conflict when:

- A pull fails with an error.

- In the Git pane, you see a status like U U

RStudio currently doesn't provide any tools to help with merge conflicts, so you'll need to use the command line. I recommend starting by setting your merge conflict "style" to `diff3`. The `diff3` style shows three things when you get a merge conflict: your local changes, the original file and the remote changes. The default style is `diff2`, which only shows your changes and the remote changes. This generally makes it harder to figure out what's happened.

- If you've encountered your first merge conflict, do the following:

  ```
  # Abort this merge
  git merge --abort
  # Set the conflict style
  git config --global merge.conflictstyle diff3
  # Re-try the merge
  git pull
  ```

- If you're not in the middle of a merge conflict, just run

  ```
  git config --global merge.conflictstyle diff3
  ```

To resolve a merge conflict, you need to open every file with the status U U. In each file, you'll find a conflict marker that looks like this:

```
<<<<<<< HEAD

||||||| merged common ancestors

=======

>>>>>>> remote
```

This shows all three versions of the conflicting code:

- At the top, your local code.

- In the middle, the code from the last commit before the split between the two lines of development (This is missing in the default conflict style, so if you don't see it, follow the instructions above).

- At the bottom, the remote code that you pulled down from GitHub.

You need to work through each conflict and decide either which version is better, or how to combine both versions. Then, before you stage the file, make sure you've deleted all the conflict markers. Once you've fixed all conflicts, make a new commit and push to GitHub.

A couple of pointers when fixing text generated by roxygen:

- Don't fix problems in `man/*.Rd` files. Instead, resolve any conflicts in the underlying roxygen comments and re-document the package.
- Merge conflicts in the `NAMESPACE` file will prevent you from re-loading or re-documenting a package. Resolve them enough so that the package can be loaded, then re-document to generate a clean and correct `NAMESPACE`.

Handling merge conflicts is one of the trickier parts of Git. You may need to read a few tutorials before you get the hang of it. Google and StackOverflow are great resources. If you get terribly confused, you can always abort the merge and try again by running `git merge --abort` then `git pull`.

13.12 Issues

Every GitHub repo comes with a page for tracking issues. Use it! If you encounter a bug while working on another project, jot a note down on the issues page. When you have a smaller project, don't worry too much about milestones, tags and assigning issues to specific people. Those are more useful once you get over a page of issues (>50). Once you get to that point, read the GitHub guide on issues: `https://guides.github.com/features/issues/`.

A useful technique is closing issues from a commit message. Just put `Closes #<issue number>` somewhere in your commit message and GitHub will close the issue for you when you next push. The best thing about closing issues this way is that it makes a link from the issue to the commit. This is useful if you ever have to come back to the bug and want to see exactly what you did to fix it. You can also link to issues without closing them; just refer to `#<issue number>`.

As you'll learn about in Section 15.5.3, it's a good idea to add a bullet to `NEWS.md` whenever you close an issue. The bullet point should describe the issue in terms that users will understand, as opposed to the commit message which is written for developers.

13.13 Branches

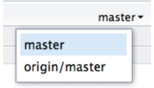
Sometimes you want to make big changes to your code without having to disturb your main stream of development. Maybe you want to break it up into multiple simple commits so you can easily track what you're doing. Maybe you're not sure what you've done is the best approach and you want someone else to review your code. Or, maybe you want to try something experimental (you can merge it back only if the experiment succeeds). Branches and pull requests provide powerful tools to handle these situations.

Although you haven't realised it, you're already using branches. The default branch is called **master**; it's where you've been saving your commits. If you synchronise your code to GitHub you'll also have a branch called **origin/master**: it's a local copy of all the commits on GitHub, which gets synchronised when you pull. `git pull` does two things:

1. `git fetch origin master` to update the local `origin/master` branch with the latest commits from GitHub.
2. `git merge orgin/master` to combine the remote changes with your changes.

It's useful to create your own branches when you want to (temporarily) break away from the main stream of development. You can create a new branch with `git checkout -b <branch-name>`. Names should be in lower case letters and numbers, with `-` used to separate words.

Switch between branches with `git checkout <branch-name>`. For example, to return to the main line of development use `git checkout master`. You can also use the branch switcher at the top right of the Git pane:



If you've forgotten the name of your branch in the shell, you can use `git branch` to list all existing branches.

If you try to synchronise this branch to GitHub from inside RStudio, you'll notice that push and pull are disabled: . To enable them, you'll need to first tell Git that your local branch has a remote equivalent:

```
git push --set-upstream origin <branch-name>
```

After you've done that once, you can use the pull and push buttons as usual.

If you've been working on a branch for a while, other work might have been going on in the master branch. To integrate that work into your branch, run `git merge master`. You will need to resolve any merge conflicts (see above). It's best to do this fairly frequently - the less your branch diverges from the master, the easier it will be to merge.

Once you're done working on a branch, merge it back into the master, then delete the branch:

```
git checkout master
git merge <branch-name>
git branch -d <branch-name>
```

(Git won't let you delete a branch unless you've merged it back into the master branch. If you do want to abandon a branch without merging it, you'll need to force delete with `-D` instead of `-d`. If you accidentally delete a branch, don't panic. It's usually possible to get it back. See the advice about undoing mistakes).

13.14 Making a pull request

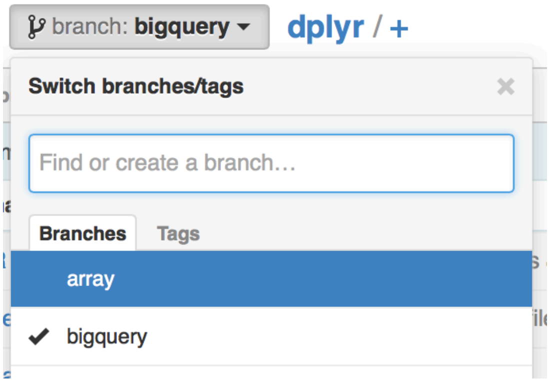
A pull request is a tool for proposing and discussing changes before merging them into a repo. The most common use for a pull request is to contribute to someone else's code: it's the easiest way to propose changes to code that you don't control.

Below, you'll learn about pull requests to make changes to your own code. This may seem a bit pointless because you don't *need* them, as you can directly modify your code. But pull requests are surprisingly useful because they allow you to get feedback on proposed changes. We use them frequently at RStudio to get feedback before merging major changes.

GitHub has some good documentation on using pull requests: `https://help.github.com/articles/using-pull-requests/`. In this chapter, I'll focus on the basics you need to know to use pull requests effectively, and show you how they fit in with the Git commands you've learned so far.

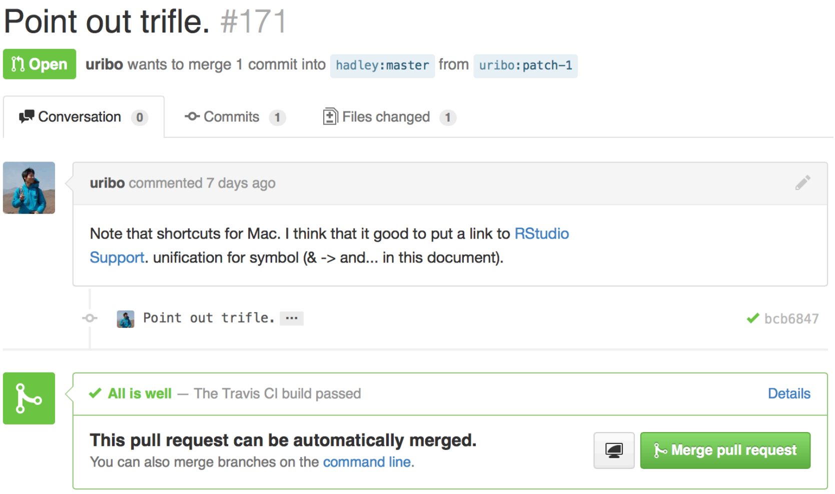
To create a pull request, you create a branch, commit code, then push the branch to GitHub. When you next go to the GitHub website you'll see a header that invites you to submit a pull request. You can also do it by:

1. Switching branches:



2. Clicking Pull Request

This will create a page that looks like this:



Point out trifle. #171

Open uribo wants to merge 1 commit into `hadley:master` from `uribo:patch-1`

Conversation 0 | Commits 1 | Files changed 1

uribo commented 7 days ago

Note that shortcuts for Mac. I think that it good to put a link to RStudio Support. unification for symbol (& -> and... in this document).

`Point out trifle.` ... ✓ bcb6847

✓ All is well — The Travis CI build passed Details

This pull request can be automatically merged.
You can also merge branches on the command line.

Merge pull request

This pull request, which fixes a couple of small problems, is one that was submitted to this book's GitHub site.

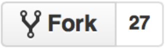
There are three parts to a pull request:

- A **conversation**, Conversation 1, where you can discuss the changes as a whole.
- The **commits** view, Commits 2, where you can see each individual commit.
- The **file changes**, Files changed 2, where you see the overall diff of the commits, and you can comment on individual lines.

Once you're done discussing a pull request, you either choose to merge it or delete it. Merging it is equivalent to running `git merge <branchname>` from the shell; deleting is equivalent to `git branch -r <branchname>`.

13.15 Submitting a pull request to another repo

To submit a pull request to a repo that you don't own, you first need to create a copy of the repo that you can own, called a **fork**, and then clone that fork on your own computer:

1. **Fork** the original repo by going to the repo on GitHub and clicking 

. This creates a copy of the repo that belongs to you.

2. **Clone** the forked repo to create a local copy of the remote repo. It's possible to do this within RStudio (using "Create New Project" from "Version Control") but I think it's easier to do it from the shell:

   ```
   git clone git@github.com:<your-name>/<repo>.git
   cd <repo>
   ```

A fork is a *static* copy of the repo: once you've created it, GitHub does nothing to keep it in sync with the upstream repo. This is a problem because while you're working on a pull request, changes might occur in the original repo. To keep the forked and the original repo in sync, start by telling your repo about the upstream repo:

```
git remote add upstream git@github.com:<original-name>/<repo>.git
git fetch upstream
```

Then you can merge changes from the upstream repo to your local copy:

```
git merge upstream/master
```

When working on a forked repo, I recommend that you don't work on the master branch. Because you're not really working on the main line of development for that repo, using your master branch makes things confusing.

If you always create pull requests in branches, you can make it a little easier to keep your local repo in sync with the upstream repo by running:

```
git branch -u upstream/master
```

Then you can update your local repo with the following code:

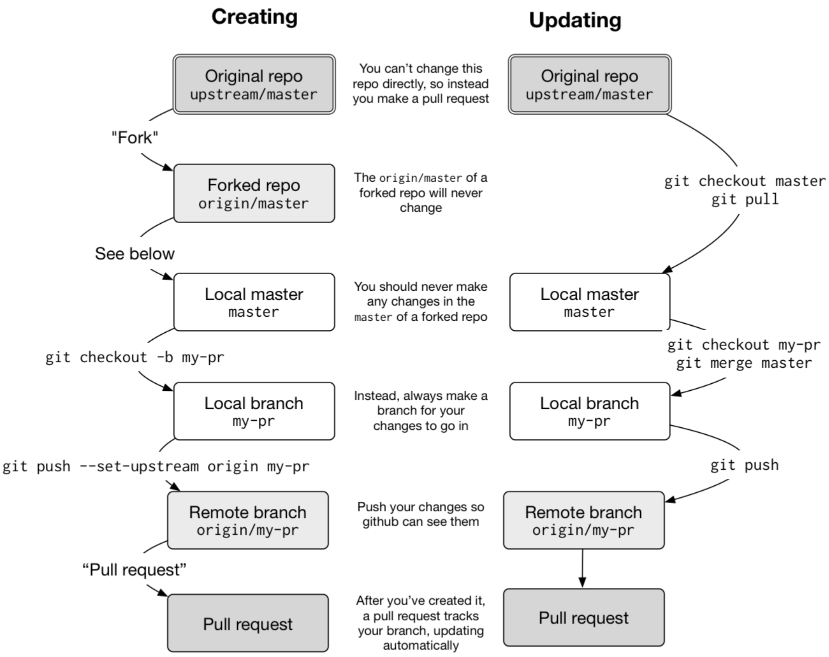
```
git checkout master
git pull
```

Changes may occur while you're working on the pull request, so remember to merge them into your branch with:

```
git checkout <my-branch>
git merge master
```

A pull request (PR) is a one-to-one mapping to a branch, so you can also use this technique to make updates based on the pull request discussion. Don't create a new pull request each time you make a change; instead you just need to push the branch that the PR is based on and the PR webpage will automatically update.

The diagram below illustrates the main steps of creating a pull request and updating the request as the upstream repo changes:


Creating
Updating
Original repo
upstream/master
You can't change this repo directly, so instead you make a pull request
Original repo
upstream/master
"Fork"
Forked repo
origin/master
The origin/master of a forked repo will never change
git checkout master
git pull
See below
Local master
master
You should never make any changes in the master of a forked repo
Local master
master
git checkout -b my-pr
git checkout my-pr
git merge master
Local branch
my-pr
Instead, always make a branch for your changes to go in
Local branch
my-pr
git push --set-upstream origin my-pr
git push
Remote branch
origin/my-pr
Push your changes so github can see them
Remote branch
origin/my-pr
"Pull request"
Pull request
After you've created it, a pull request tracks your branch, updating automatically
Pull request

```
# Initial setup
git clone git@github.com:<your-name>/extrafont.git
git remote add upstream git@github.com:<original-name>/extrafont.git
git fetch upstream
git branch -u upstream/master
```

13.16 Reviewing and accepting pull requests

As your package gets more popular, you're likely to receive pull requests. Receiving a pull request is fantastic. Someone not only cares about your package enough to use it, they've actually read the source code and made an improvement!

When you receive a pull request, I recommend reviewing it using the three step approach described by Sarah Sharp. I summarise the three phases below, but I highly recommend reading the full article at http://sarah.thesharps.us/2014/09/01/the-gentle-art-of-patch-review/:

1. Is it a good idea? If you don't think the contribution is a good fit for your project, it's polite to let the contributor know as quickly as possible. Thank them for their work, and refocus them on a better area to work on.

2. Is the overall approach sound? At this point you want to focus on the big picture: have they modified the right functions in the right way? Avoid nitpicking minor style problems (that's the final phase); instead just provide a pointer to your style preferences, e.g. http://r-pkgs.had.co.nz/style.html.

3. Is it polished? In the final review phase, make sure that the non-code parts of the PR are polished. Prompt the contributor to update the documentation, point out spelling mistakes and suggest better wording.

   I recommend asking the contributor to include a bullet point in NEWS.md, briefly describing the improvement and thanking themselves with their GitHub username. More details to follow in Section 15.7).

After discussion is complete, you can incorporate the changes by clicking the merge button. If the button doesn't work, GitHub provides some instructions on how to do it from the command line. While you've seen all the pieces before, it's useful to read through this just so you understand what exactly is happening.

```
# Create a new branch, and sync it with the pull request
git checkout -b <branch> master
git pull https://github.com/<user>/<branch>.git patch-3

# Merge the changes into the main line of development
git checkout master
git merge --no-ff <branch>
# Resolve conflicts, stage and add.

# Sync your local changes with GitHub
git push origin master
```

13.17 Learning more

Git and GitHub are a rich and powerful set of tools, and there's no way this chapter has taught you everything you need to know. However, you should now have the basic knowledge to be effective, and you should be in a good position to learn more. Some good resources are:

- GitHub help, https://help.github.com, not only teaches you about GitHub, but also has good tutorials on many Git features.

- If you'd like to learn more about the details of Git, read Pro Git (`http://git-scm.com/book/en/v2`) by Scott Chacon and Ben Straub.

Finally, StackOverflow is a vital part of Git - when you have a problem that you don't know how to solve, StackOverflow should be your first resource. It's highly likely that someone has already had the same exact problem as you, and that there will be a variety of approaches and solutions to choose from.

Chapter 14

Automated checking

An important part of the package development process is `R CMD check`. `R CMD check` automatically checks your code for common problems. It's essential if you're planning on submitting to CRAN, but it's useful even if you're not because it automatically detects many commons problems that you'd otherwise discover the hard way.

`R CMD check` will be frustrating the first time you run it - you'll discover many problems that need to be fixed. The key to making `R CMD check` less frustrating is to actually run it more often: the sooner you find a problem, the easier it is to fix. The upper limit of this approach is to run `R CMD check` every time you make a change. If you use GitHub, you'll learn precisely how to do that with Section 14.3.

14.1 Workflow

`R CMD check` is the name of the command you run from the terminal. I don't recommend calling it directly. Instead, run `devtools::check()`, or press Cmd + Shift + E in RStudio. In contrast to `R CMD check`, `devtools::check()`:

- Ensures that the documentation is up-to-date by running `devtools::document()`.
- Bundles the package before checking it. This is the best practice for checking packages because it makes sure the check starts with a clean slate: because a package bundle doesn't contain any of the temporary files that can accumulate in your source package, e.g. artifacts like `.so` and `.o` files which accompany compiled code, you can avoid the spurious warnings such files will generate.

- Sets the `NOT_CRAN` environment variable to `TRUE`. This allows you to selectively skip tests on CRAN. (See `?testthat::skip_on_cran` for details.)

The workflow for checking a package is simple, but tedious:

1. Run `devtools::check()`, or press Ctrl/Cmd + Shift + E.
2. Fix the first problem.
3. Repeat until there are no more problems.

`R CMD check` returns three types of messages:

- `ERROR`s: Severe problems that you should fix regardless of whether or not you're submitting to CRAN.
- `WARNING`s: Likely problems that you must fix if you're planning to submit to CRAN (and a good idea to look into even if you're not).
- `NOTE`s: Mild problems. If you are submitting to CRAN, you should strive to eliminate all NOTEs, even if they are false positives. If you have no NOTEs, human intervention is not required, and the package submission process will be easier. If it's not possible to eliminate a `NOTE`, you'll need describe why it's OK in your submission comments, as described in Section 15.3.2. If you're not submitting to CRAN, carefully read each NOTE, but don't go out of your way to fix things that you don't think are problems.

14.2 Checks

`R CMD check` is composed of over 50 individual checks, described in the following sections. For each check, I briefly describe what it does, what the most common problems are, and how to fix them. When you have a problem with `R CMD check` and can't understand how to fix it, use this list to help figure out what you need to do. To make it easier to understand how the checks fit together, I've organised them into sections roughly corresponding to the chapters in this book. This means they will be in a somewhat different order to what you'll see when you run `check()`.

This list includes every check run in R 3.1.1. If you're using a more recent version, you may want to consult the most recent online version of this chapter: `http://r-pkgs.had.co.nz/check.html`. Please let me know (`mailto:hadley@rstudio.com`) if you encounter a problem that this chapter doesn't help with.

14.2.1 Check metadata

`R CMD check` always starts by describing your current environment. I'm running R 3.1.1 on OS X with a UTF-8 charset:

- **Using log directory** '/Users/hadley/Documents/web/httr.Rcheck'
- **Using R version** 3.1.1 (2014-07-10)
- **Using platform**: x86_64-apple-darwin13.1.0 (64-bit)
- **Using session charset**: UTF-8

Next the description is parsed and the package version is printed. Here I'm checking httr version 0.5.0.9000 (you'll learn more about that weird version number in Section 4.5).

- **Checking for file** 'httr/DESCRIPTION'
- **This is package** 'httr' **version** '0.5.0.9000'

14.2.2 Package structure

- **Checking package directory**. The directory you're checking must exist - `devtools::check()` protects you against this problem.

- **Checking if this is a source package**. You must check a source package, not a binary or installed package. This should never fail if you use `devtools::check()`.

- **Checking for executable files**. You must not have executable files in your package: they're not portable, they're not open source, and they are a security risk. Delete any executable files from your package. (If you're not submitting to CRAN, you can silence this warning by listing each executable file in the `BinaryFiles` field in your `DESCRIPTION`.)

- **Checking for hidden files and directories**. On Linux and OS X, files with a name starting with `.` are hidden by default, and you've probably included them in your package by mistake. Either delete them, or if they are important, use `.Rbuildignore` to remove them from the package bundle. R automatically removes some common directories like `.git` and `.svn`.

- **Checking for portable file names**. R packages must work on Windows, Linux and OS X, so you can only use file names that work on all platforms. The easiest way to do this is to stick to letters, numbers, underscores and dashes. Avoid non-English letters and spaces. Fix this check by renaming the listed files.

- **Checking for sufficient/correct file permissions**. If you can't read a file, you can't check it. This check detects the unlikely occurence that you have files in the package that you don't have permission to read. Fix this problem by fixing the file permissions.

- **Checking whether package 'XYZ' can be installed**. `R CMD check` runs `R CMD install` to make sure that it's possible to install your package. If this fails, you should run `devtools::install()` or RStudio's Build & Reload and debug any problems before continuing.

- **Checking installed package size**. It's easy to accidentally include large files that blow up the size of your package. This check ensures that the whole package is less than 5 MB and each subdirectory is less than 1 MB. If you see this message, check that you haven't accidentally included a large file.

  If submitting to CRAN, you'll need to justify the size of your package. First, make sure the package is as small as it possibly can be: try recompressing the data, Section 9.5; and minimising vignettes, Section 6.7. If it's still too large, consider moving data into its own package.

- **Checking top-level files**. Only specified files and directories are allowed at the top level of the package (e.g. `DESCRIPTION`, `R/`, `src/`). To include other files, you have two choices:

  - If they don't need to be installed (i.e. they're only used in the source package): add them to `.Rbuildignore` with `devtools::use_build_ignore()`.
  - If they need to be installed: move them into `inst/`. They'll be moved back to the top-level package directory when installed.

- **Checking package subdirectories**.

  - Don't include any empty directories. These are usually removed automatically by `R CMD build` so you shouldn't see this error. If you do, just delete the directory.
  - The case of files and directories is important. All sub-directories should be lower-case, except for `R/`. A citation file, if present, should be in `inst/CITATION`. Rename as needed.
  - The contents of `inst/` shouldn't clash with top-level contents of the package (like `build/`, `R/` etc). If they do, rename your files/directories.

- **Checking for left-over files**. Remove any files listed here. They've been included in your package by accident.

14.2.3 Description

- **Checking DESCRIPTION meta-information**.
  - The `DESCRIPTION` must be valid. You are unlikely to see this error, because `devtools::load_all()` runs the same check each time you re-load the package.
  - If you use any non-ASCII characters in the DESCRIPTION, you must also specify an encoding. There are only three encodings that work on all platforms: latin1, latin2 and UTF-8. I strongly recommend UTF-8: `Encoding: UTF-8`.
  - The `License` must refer to either a known license (a complete list can be found at `https://svn.r-project.org/R/trunk/share/licenses/license.db`), or it must use `file LICENSE` and that file must exist. Errors here are most likely to be typos.
  - You should either provide `Authors@R` or `Authors` and `Maintainer`. You'll get an error if you've specified both, which you can fix by removing the one you didn't want.

- **Checking package dependencies**.
  - All packages listed in `Depends`, `Imports` and `LinkingTo` must be installed, and their version requirements must be met, otherwise your package can't be checked. An easy way to install any missing or outdated dependencies is to run `devtools::install_deps(dependencies = TRUE)`.
  - Packages listed in `Suggests` must be installed, unless you've set the environment variable `_R_CHECK_FORCE_SUGGESTS_` to a false value (e.g. with `check(force_suggests = FALSE)`). This is useful if some of the suggested packages are not available on all platforms.
  - R packages can not have a cycle of dependencies: i.e. if package A requires B, then B can not require A (otherwise which one would you load first?). If you see this error, you'll need to rethink the design of your package. One easy fix is to move the conflicting package from `Imports` or `Depends` to `Suggests`.
  - Any packages used in the `NAMESPACE` must be listed in one of `Imports` (most commonly) or `Depends` (only in special cases). See Section 8.2 for more details.
  - Every package listed in `Depends` must also be imported in the `NAMESPACE` or accessed with `pkg::foo`. If you don't do this, your package will work when attached to the search path (with `library(mypackage)`) but will not work when only loaded (e.g. `mypackage::foo()`)

- **Checking CRAN incoming feasibility**. These checks only apply if you're submitting to CRAN.
  - If you're submitting a new package, you can't use the same name as an existing package. You'll need to come up with a new name.
  - If you're submitting an update, the version number must be higher than the current CRAN version. Update the `Version` field in `DESCRIPTION`.
  - If the maintainer of the package has changed (even if it's just a change in email address), the new maintainer should submit to CRAN, and the old maintainer should send a confirmation email.
  - You must use a standard open source license, as listed in `https://svn.r-project.org/R/trunk/share/licenses/license.db`. You can not use a custom license as CRAN does not have the legal resources to review custom agreements.
  - The `Title` and `Description` must be free from spelling mistakes. The title of the package must be in title case. Neither title nor description should include either the name of your package or the word "package". Reword your title and description as needed.
  - If you're submitting a new package, you'll always get a `NOTE`. This reminds the CRAN maintainers to do some extra manual checks.
  - Avoid submitting multiple versions of the same package in a short period of time. CRAN prefers at most one submission per month. If you need to fix a major bug, be apologetic.

14.2.4 Namespace

- **Checking if there is a namespace**. You must have a `NAMESPACE` file. Roxygen2 will create this for you as described in Chapter 8.

- **Checking package namespace information**. The `NAMESPACE` should be parseable by `parseNamespaceFile()` and valid. If this check fails, it's a bug in roxygen2.

- **Checking whether the package can be loaded with stated dependencies**. Runs `library(pkg)` with `R_DEFAULT_PACKAGES=NULL`, so the search path is empty (i.e. stats, graphics, grDevices, utils, datasets and methods are not attached like usual). Failure here typically indicates that you're missing a dependency on one of those packages.

- **Checking whether the namespace can be loaded with stated dependencies**. Runs `loadNamespace(pkg)` with `R_DEFAULT_PACKAGES=NULL`. Failure usually indicates a problem with the namespace.

14.2.5 R code

- **Checking R files for non-ASCII characters**. For maximum portability (i.e. so people can use your package on Windows) you should avoid using non-ASCII characters in R files. It's ok to use them in comments, but object names shouldn't use them, and in strings you should use unicode escapes. See Section 3.5 for more details.

- **Checking R files for syntax errors**. Obviously your R code must be valid. You're unlikely to see this error if you're been regularly using `devtools::load_all()`.

- **Checking dependencies in R code**. Errors here often indicate that you've forgotten to declare a needed package in the `DESCRIPTION`. Remember that you should never use `require()` or `library()` inside a package - see Section 8.6 for more details on best practices.

  Alternatively, you may have accidentally used `:::` to access an exported function from a package. Switch to `::` instead.

- **Checking S3 generic/method consistency**. S3 methods must have a compatible function signature with their generic. This means that the method must have the same arguments as its generic, with one exception: if the generic includes `...` the method can have additional arguments.

  A common cause of this error is defining print methods, because the `print()` generic contains`...`:

  ```
  # BAD
  print.my_class <- function(x) cat("Hi")

  # GOOD
  print.my_class <- function(x, ...) cat("Hi")

  # Also ok
  print.my_class <- function(x, ..., my_arg = TRUE) cat("Hi")
  ```

- **Checking replacement functions**. Replacement functions (e.g. functions that are called like `foo(x) <- y`), must have `value` as the last argument.

- **Checking R code for possible problems**. This is a compound check for a wide range of problems:

  - Calls to `library.dynam()` (and `library.dynam.unload()`) should look like `library.dynam("name")`, not `library.dynam("name.dll")`. Remove the extension to fix this error.

- Put `library.dynam()` in `.onLoad()`, not `.onAttach()`; put `packageStartupMessage()` in `.onAttach()`, not `.onLoad()`. Put `library.dynam.unload()` in `.onUnload()`. If you use any of these functions, make sure they're in the right place.
- Don't use `unlockBinding()` or `assignInNamespace()` to modify objects that don't belong to you.
- `codetools::checkUsagePackage()` is called to check that your functions don't use variables that don't exist. This sometimes raises false positives with functions that use non-standard evaluation (NSE), like `subset()` or `with()`. Generally, I think you should avoid NSE in package functions, and hence avoid this NOTE, but if you can not, see `?globalVariables` for how to suppress this NOTE.
- You are not allowed to use `.Internal()` in a package. Either call the R wrapper function, or write your own C function. (If you copy and paste the C function from base R, make sure to maintain the copyright notice, use a GPL-2 compatible license, and list R-core in the `Author` field.)
- Similarly you are not allowed to use `:::` to access non-exported functions from other packages. Either ask the package maintainer to export the function you need, or write your own version of it using exported functions. Alternatively, if the licenses are compatible you can copy and paste the exported function into your own package. If you do this, remember to update `Authors@R`.
- Don't use `assign()` to modify objects in the global environment. If you need to maintain state across function calls, create your own environment with `e <- new.env(parent = emptyenv())` and set and get values in it:

```r
e <- new.env(parent = emptyenv())

add_up <- function(x) {
  if (is.null(e$last_x)) {
    old <- 0
  } else {
    old <- e$last_x
  }

  new <- old + x
  e$last_x <- new
  new
}
add_up(10)
#> [1] 10
add_up(20)
#> [1] 30
```

  - Don’t use `attach()` in your code. Instead refer to variables explicitly.
  - Don’t use `data()` without specifying the `envir` argument. Otherwise the data will be loaded in the global environment.
  - Don’t use deprecated or defunct functions. Update your code to use the latest versions.
  - You must use `TRUE` and `FALSE` in your code (and examples), not `T` and `F`.

- **Checking whether the package can be loaded**. R loads your package with `library()`. Failure here typically indicates a problem with `.onLoad()` or `.onAttach()`.

- **Checking whether the package can be unloaded cleanly**. Loads with `library()` and then `detach()`es. If this fails, check `.onUnload()` and `.onDetach()`.

- **Checking whether the namespace can be unloaded cleanly**. Runs `loadNamespace("pkg"); unloadNamespace("pkg")`. Check `.onUnload()` for problems.

- **Checking loading without being on the library search path**. Calls `library(x, lib.loc = ...)`. Failure here indicates that you are making a false assumption in `.onLoad()` or `.onAttach()`.

14.2.6 Data

- **Checking contents of ‘data’ directory**.

  - The data directory can only contain file types described in Section 9.1.
  - Data files can only contain non-ASCII characters if the encoding is not correctly set. This usually shouldn’t be a problem if you’re saving `.Rdata` files. If you do see this error, look at the `Encoding()` of each column in the data frame, and ensure none are “unknown”. (You’ll typically need to fix this somewhere in the import process).
  - If you’ve compressed a data file with `bzip2` or `xz` you need to declare at least `Depends: R (>= 2.10)` in your `DESCRIPTION`.
  - If you’ve used a sub-optiomal compression algorithm for your data, re-compress with the suggested algorithm.

14.2.7 Documentation

You can run the most common of these outside `devtools::check()` with `devtools::check_doc()` (which automatically calls `devtools::document()` for you). If you have documentation problems, it's best to iterate quickly with `check_doc()`, rather than running the full check each time.

- **Checking Rd files**. This checks that all `man/*.Rd` files use the correct Rd syntax. If this fails, it indicates a bug in roxygen2.

- **Checking Rd metadata**. Names and aliases must be unique across all documentation files in a package. If you encounter this problem you've accidentally used the same `@name` or `@aliases` in multiple places; make sure they're unique.

- **Checking Rd line widths**. Lines in Rd files must be less than 90 characters wide. This is unlikely to occur if you wrap your R code, and hence roxygen comments, to 80 characters. For very long urls, use a link-shortening service like bit.ly (`http://bit.ly`).

- **Checking Rd cross-references**. Errors here usually represent typos. Recall the syntax for linking to functions in other packages: `\link[package_name]{function_name}`. Sometimes I accidentally switch the order of `\code{}` and `\link{}`: `\link{\code{function}}` will not work.

- **Checking for missing documentation entries**. All exported objects must be documented. See `?tools::undoc` for more details.

- **Checking for code/documentation mismatches**. This check ensures that the documentation matches the code. This should never fail because you're using roxygen2 which automatically keeps them in sync.

- **Checking Rd `\usage` sections**. All arguments must be documented, and all `@params` must document an existing argument. You may have forgotten to document an argument, forgotten to remove the documentation for an argument that you've removed, or misspelled an argument name.

  S3 and S4 methods need to use special `\S3method{}` and `\S4method{}` markup in the Rd file. Roxygen2 will generate this for you automatically.

- **Checking Rd contents**. This checks for autogenerated content made by `package.skeleton()`. Since you're not using `package.skeleton()` you should never have a problem here.

- **Checking for unstated dependencies in examples**. If you use a package only for an example, make sure it's listed in the `Suggests` field. Before running example code that depends on it, test to see if it's available with `requireNamespace("pkg", quietly = TRUE)`:

  ```
  #' @examples
  #' if (requireNamespace("dplyr", quietly = TRUE)) {
  #'   ...
  #' }
  ```

- **Checking examples**. Every documentation example must run without errors, and must not take too long. Exclude failing or slow tests with `\donttest{}`. See Section 5.4 for more details.

  Examples are one of the last checks run, so fixing problems can be painful if you have to run `devtools::check()` each time. Instead, use `devtools::run_examples()`: it only checks the examples, and 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 it.

  NB: you can't use unexported functions and you shouldn't open new graphics devices or use more than two cores. Individual examples shouldn't take more than 5s.

- **Checking PDF version of manual**. Occassionally you'll get an error when building the PDF manual. This is usually because the pdf is built by latex and you've forgotten to escape something. Debugging this is painful - your best bet is to look up the latex logs and combined tex file and work back from there to `.Rd` files then back to a roxygen comment. I consider any such failure to be a bug in roxygen2, so please let me know.

14.2.8 Demos

- **Checking index information**. If you've written demos, each demo must be listed in `demos/00Index`. The file should look like:

  ```
  demo-name-without-extension  Demo description
  another-demo-name            Another description
  ```

14.2.9 Compiled code

- **Checking foreign function calls**. `.Call()`, `.C()`, `.Fortran()`, `.External()` must always be called either with a `NativeSymbolInfo` object (as created with `@useDynLib`) or use the `.package` argument. See `?tools::checkFF` for more details.

- **Checking line endings in C/C++/Fortran sources/headers**. Always use LF as a line ending.

- **Checking line endings in Makefiles**. As above.

- **Checking for portable use of `$(BLAS_LIBS)` and `$(LAPACK_LIBS)`**. Errors here indicate an issue with your use of BLAS and LAPACK.

- **Checking compiled code**. Checks that you're not using any C functions that you shouldn't. See details in Section 10.2.6.

14.2.10 Tests

- **Checking for unstated dependencies in tests**. Every package used by tests must be included in the dependencies.

- **Checking tests**. Each file in `tests/` is run. If you've followed the instructions in Chapter 7 you'll have at least one file: `testthat.R`. The output from `R CMD check` is not usually that helpful, so you may need to look at the logfile `package.Rcheck/tests/testthat.Rout`. Fix any failing tests by iterating with `devtools::test()`.

  Occasionally you may have a problem where the tests pass when run interactively with `devtools::test()`, but fail when in `R CMD check`. This usually indicates that you've made a faulty assumption about the testing environment, and it's often hard to figure it out.

14.2.11 Vignettes

- **Checking 'build' directory**. `build/` is used to track vignette builds. I'm not sure how this check could fail unless you've accidentally `.Rbuildignored` the `build/` directory.

- **Checking installed files from 'inst/doc'**. Don't put files in `inst/doc` - vignettes now live in `vignettes/`.

- **Checking files in 'vignettes'**. Problems here are usually straightforward - you've included files that are already included in R (like `jss.cls`, `jss.bst`, or `Sweave.sty`), or you have leftover latex compilation files. Delete these files.

- **Checking for sizes of PDF files under 'inst/doc'**. If you're making PDF vignettes, you can make them as small as possible by running `tools::compactPDF()`.

- **Checking for unstated dependencies in vignettes**. As with tests, every package that you use in a vignette must be listed in the `DESCRIPTION`. If a package is used only for a vignette, and not elsewhere, make sure it's listed in `Suggests`.

- **Checking package vignettes in 'inst/doc'**. This checks that every source vignette (i.e. `.Rmd`) has a built equivalent (i.e. `.html`) in `inst/doc`. This shouldn't fail if you've used the standard process outlined in Chapter 6. If there is a problem, start by checking your `.Rbuildignore`.

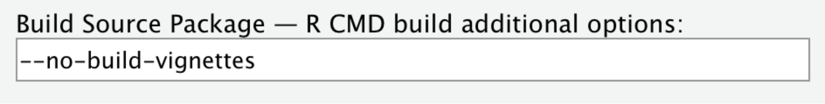
- **Checking running R code from vignettes**. The R code from each vignette is run. If you want to deliberately execute errors (to show the user what failure looks like), make sure the chunk has `error = TRUE, purl = FALSE`.

- **Checking re-building of vignette outputs**. Each vignette is re-knit to make sure that the output corresponds to the input. Again, this shouldn't fail in normal circumstances.

To run vignettes, the package first must be installed. That means `check()`:

1. Builds the package.
2. Installs the package without vignettes.
3. Builds all the vignettes.
4. Re-installs the package with vignettes.

If you have a lot of compiled code, this can be rather slow. You may want to add `--no-build-vignettes` to the commands list in "Build Source Packages" field in the project options:



Build Source Package — R CMD build additional options:

```
--no-build-vignettes
```

14.3 Checking after every commit with Travis

If you use git and GitHub, as described in Chapter 13, I highly recommend learning about Travis (`https://travis-ci.org/`). Travis is a continuous integration service, which means that it runs automated testing code everytime you push to GitHub. For open source projects, Travis provides 50 minutes of free computation on a ubuntu server for every push. For an R package, the most useful code to run is `devtools::check()`.

To use Travis:

1. Run `devtools::use_travis()` to set up a basic `.travis.yml` config file.
2. Navigate to your Travis account (`https://travis-ci.org/profile`) and enable Travis for the repo you want to test.
3. Commit and push to GitHub.
4. Wait a few minutes to see the results in your email.

With this setup in place, every time you push to GitHub, and every time someone submits a pull request, `devtools::check()` will be automatically run. You'll find out about failures right away, which makes them easier to fix. Using Travis also encourages me to check more often locally, because I know if it fails I'll find out about it a few minutes later, often once I've moved on to a new problem.

14.3.1 Basic config

The Travis config is stored in a yaml file called `.travis.yml`. The default config created by devtools looks like this:

```
language: r
warnings_are_errors: true
sudo: required
```

R has recently become a community supported language on Travis and you can read the documentation at `http://docs.travis-ci.com/user/languages/r/`.

There are two particularly useful options:

`r_github_packages` : A list of R packages to install from github. This allows you to test against the development version of your depedencies.

`r_binary_packages` : A list of precompiled R packages to install from ubuntu. This allows you to reduce your the build time. You can see if a binary version of a package is available by searching on `http://packages.ubuntu.com` for `r-cran-lowercasename`. For example, searching for `r-cran-xml` reveals that you can get a binary version of XML package.

14.3.2 Other uses

Since Travis allows you to run arbitrary code, there are many other things that you can use it for:

- Re-publishing a book website every time you make a change to the source. (Like this book!)

- Building vignettes and publishing them to a website.
- Automatically building a documentation website for your package.

To learn more, read about the many deployment options (`http://docs.travis-ci.com/user/deployment/`) provided by Travis.

Chapter 15

Releasing a package

If you want your package to have significant traction in the R community, you need to submit it to CRAN. Submitting to CRAN is a lot more work than just providing a version on github, but the vast majority of R users do not install packages from github, because CRAN provides discoverability, ease of installation and a stamp of authenticity. The CRAN submission process can be frustrating, but it's worthwhile, and this chapter will make it as painless as possible.

To get your package ready to release, follow these steps:

1. Pick a version number.
2. Run and document `R CMD check`.
3. Check that you're aligned with CRAN policies.
4. Update `README.md` and `NEWS.md`.
5. Submit the package to CRAN.
6. Prepare for the next version by updating version numbers.
7. Publicise the new version.

15.1 Version number

If you've been following the advice in Section 4.5, the version number of your in-development package will have four components, `major.minor.patch.dev`, where `dev` is at least 9000. The number 9000 is arbitrary, but provides a strong visual signal there's something different about this version number. Released packages don't have a `dev` component, so now you need to drop that and pick a version number based on the changes you've made. For example, if the current version is `0.8.1.9000` will the next CRAN version be `0.8.2`, `0.9.0` or `1.0.0`? Use this advice to decide:

- Increment `patch`, e.g. `0.8.2` for a **patch**: you've fixed bugs without adding any significant new features. I'll often do a patch release if, after release, I discover a show-stopping bug that needs to be fixed ASAP. Most releases will have a patch number of 0.

- Increment `minor`, e.g. `0.9.0`, for a **minor release**. A minor release can include bug fixes, new features and changes in backward compatibility. This is the most common type of release. It's perfectly fine to have so many minor releases that you need to use two (or even three!) digits, e.g. `1.17.0`.

- Increment `major`, e.g. `1.0.0`, for a **major release**. This is best reserved for changes that are not backward compatible and that are likely to affect many users. Going from `0.b.c` to `1.0.0` typically indicates that your package is feature complete with a stable API.

  In practice, backward compatibility is not an all-or-nothing threshold. 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. But if you fix a bug that many people depend on, it will feel like an API breaking change. Use your best judgement.

15.2 Backward compatibility

The big difference between major and minor versions is whether or not the code is backward compatible. This difference is a bit academic in the R community because the way most people update packages is by running `update.packages()`, which always updates to the latest version of the package, even if the major version has changed, potentially breaking code. While more R users are becoming familiar with tools like packrat (`http://rstudio.github.io/packrat/`), which capture package versions on a per-project basis, you do need to be a little cautious when making big backward incompatible changes, regardless of what you do with the version number.

The importance of backward compatibility is directly proportional to the number of people using your package: you are trading your time for your users' time. The harder you strive to maintain backward compatibility, the harder it is to develop new features or fix old mistakes. Backward compatible code also tends to be harder to read because of the need to maintain multiple paths to support functionality from previous versions. Be concerned about backward compatibility, but don't let it paralyse you.

There are good reasons to make backward incompatible changes - if you made a design mistake that makes your package harder to use it's better to fix it sooner rather than later. If you do need to make a backward incompatible change, it's best to do it gradually. Provide interim version(s) between where are you now and where you'd like to be, and provide advice about what's going to change.

Depending on what you're changing, use one of the following techniques to let your users know what's happening:

- Don't immediately remove a function. First deprecate it. For example, imagine your package is version `0.5.0` and you want to remove `fun()`. In version, `0.6.0`, you'd use `.Deprecated()` to display a warning message whenever someone uses the function:

  ```
  # 0.1.0
  fun <- function(x, y, z) {
    .Deprecated("sum")
    x + y + z
  }

  fun(1, 2, 3)
  #> Warning: 'fun' is deprecated.
  #> Use 'sum' instead.
  #> See help("Deprecated")
  #> [1] 6
  ```

  Then, remove the function once you got to `0.7.0` (or if you are being very strict, once you got to `1.0.0` since it's a backward incompatible change).

- Similarly, if you're removing a function argument, first warn about it:

  ```
  bar <- function(x, y, z) {
    if (!missing(y)) {
      warning("argument y is deprecated; please use z instead.",
        call. = FALSE)
      z <- y
    }
  }

  bar(1, 2, 3)
  #> Warning: argument y is deprecated; please use z instead.
  ```

- If you're deprecating a lot of code, it can be useful to add a helper function. For example, ggplot2 has `gg_dep` which automatically displays a message, warning or error, depending on how much the version number has changed.

  ```
  gg_dep <- function(version, msg) {
    v <- as.package_version(version)
    cv <- packageVersion("ggplot2")

    # If current major number is greater than last-good major number, or if
    # current minor number is more than 1 greater than last-good minor number,
    # return an error.
  ```

```
  if (cv[[1,1]] > v[[1,1]]  ||  cv[[1,2]] > v[[1,2]] + 1) {
    stop(msg, " (Defunct; last used in version ", version, ")",
      call. = FALSE)

  # If minor number differs by one, give a warning
  } else if (cv[[1,2]] > v[[1,2]]) {
    warning(msg, " (Deprecated; last used in version ", version, ")",
      call. = FALSE)

  # If only subminor number is greater, provide a message
  } else if (cv[[1,3]] > v[[1,3]]) {
    message(msg, " (Deprecated; last used in version ", version, ")")
  }

  invisible()
}
```

- Significant changes to an existing function requires planning, including making gradual changes over multiple versions. Try and develop a sequence of transformations where each change can be accompanied by an informative error message.

- If you want to use functionality in a new version of another package, don't make it a hard install-time dependency in the `DESCRIPTION` (forcing your users to upgrade that package might break other code). Instead check for the version at run-time:

  ```
  if (packageVersion("ggplot2") < "1.0.0") {
    stop("ggplot2 >= 1.0.0 needed for this function.", call. = FALSE)
  }
  ```

  This is also useful if you're responding to changes in one of your dependencies - you'll want to have a version that will work both before and after the change. This will allow you to submit it to CRAN at any time, even before the other package. Doing this may generate some `R CMD check` notes. For example:

  ```
  if (packageVersion("foo" > "1.0.0")) {
    foo::baz()
  } else {
    foo::bar()
  }
  ```

  If `baz` doesn't exist in foo version 1.0.0, you'll get a note that it doesn't exist in foo's namespace. Just explain that you're working around a difference between versions in your submission to CRAN.

15.3 The submission process

To manually submit your package to CRAN, you create a package bundle (with `devtools::build()`) then upload it to `http://cran.r-project.org/submit.html`, along with some comments which describe the process you followed. This section shows you how to make submission as easy as possible by providing a standard structure for those comments. Later, in Section 15.6, you'll see how to actually submit the package with `devtools::release()`.

When submitting to CRAN, remember that CRAN is staffed by volunteers, all of whom have other full-time jobs. In a typical week, there over 100 submissions and only three volunteers to process them all. The less work you make for them the more likely you are to have a pleasant submission experience.

I recommend that you store your submission comments in a file called `cran-comments.md`. `cran-comments.md` should be checked into git (so you can track it over time), and listed in `.Rbuildignore` (so it's not included in the package). As the extension suggests, I recommend using Markdown because it gives a standard way of laying out plain text. However, because the contents will never be rendered to another format, you don't need to worry about sticking to it too closely. Here are the `cran-comments.md` from a recent version of httr:

```
## Test environments
* local OS X install, R 3.1.2
* ubuntu 12.04 (on travis-ci), R 3.1.2
* win-builder (devel and release)

## R CMD check results
There were no ERRORs or WARNINGs.

There was 1 NOTE:

* checking dependencies in R code ... NOTE
  Namespace in Imports field not imported from: 'R6'

  R6 is a build-time dependency.

## Downstream dependencies
I have also run R CMD check on downstream dependencies of httr
(https://github.com/wch/checkresults/blob/master/httr/r-release).
All packages that I could install passed except:

* Ecoengine: this appears to be a failure related to config on
  that machine. I couldn't reproduce it locally, and it doesn't
  seem to be related to changes in httr (the same problem exists
  with httr 0.4).
```

This layout is designed to be easy to skim, and easy to match up to the `R CMD check` results seen by CRAN maintainers. It includes three sections:

1. Test environments: this describes where I checked the package. I always check on three platforms: my mac, Section 14.3 and win-builder.
2. Check results: I always state that there were no errors or warnings. Any `NOTE`s go in a bulleted list. For each `NOTE`, I include the message from `R CMD check` and a brief description of why I think it's OK. If there were no `NOTE`s, I'd say "There were no ERRORs, WARNINGs or NOTEs"
3. Downstream dependencies: If there are downstream dependencies, I run `R CMD check` on each package and summarise the results. If there are no downstream dependencies, keep this section, but say: "There are currently no downstream dependencies for this package".

These are described in more detail below.

15.3.1 Test environments

When checking your package you need to make sure that it passed with the current development version of R and it works on at least two platforms. `R CMD check` is continuously evolving, so it's a good idea to check your package with the latest development version, **R-devel**. You can install R-devel on your own machine:

- Mac: install from `http://r.research.att.com`.
- Windows: install from `http://cran.r-project.org/bin/windows/base/rdevel.html`
- Linux: either build it from source, or better, learn about Docker containers and run the R-devel container from `https://github.com/rocker-org/rocker`.

It's painful to manage multiple R versions, especially since you'll need to reinstall all your packages. Instead, you can run `R CMD check` on CRAN's servers with `devtools::build_win()`. This builds your package and submits it to the CRAN win-builder. 10-20 minutes after submission, you'll receive an e-mail telling you the check results.

CRAN runs on multiple platforms: Windows, Mac OS X, Linux and Solaris. You don't need to run `R CMD check` on every one of these platforms, but it's a really good idea to do it on at least two. This increases your chances of spotting code that relies on the idiosyncrasies of specific platform. If you're on linux or the mac, use `devtools::build_win()` to check on windows. If you're on windows, use Travis, as described in Section 14.3, to run checks on linux.

Debugging code that works on your computer but fails elsewhere is painful. If that happens to you, either install a virtualisation tool so that you can run another operating system locally, or find a friend to help you figure out the problem. Don't submit the package and hope CRAN will help you figure out the problem.

15.3.2 Check results

You've already learned how to use `R CMD check` and why it's important in Chapter 14. Compared to running `R CMD check` locally, there are a few important differences when running it for a CRAN submission:

- You must fix all `ERROR`s and `WARNING`s. A package that contains any errors or warnings will not be accepted by CRAN.
- Eliminate as many `NOTE`s as possible. Each `NOTE` requires human oversight, which is a precious commodity. If there are notes that you do not believe are important, it is almost always easier to fix them (even if the fix is a bit of a hack) than to persuade CRAN that they're OK. See Section 14.2 for details on how to fix individual problems.

  If you have no `NOTE`s it is less likely that your package will be flagged for additional human checks. These are time consuming for both you and CRAN, so are best avoided if possible.
- If you can't eliminate a `NOTE`, document it in `cran-comments.md`, describing why you think it is spurious. Your comments should be easy to scan, and easy to match up with `R CMD check`. Provide the CRAN maintainers with everything they need in one place, even if it means repeating yourself.

  NB: There will always be one `NOTE` when you first submit your package. This reminds CRAN that this is a new submission and that they'll need to do some extra checks. You can't eliminate this, so just mention in `cran-comments.md` that this is your first submission.

15.3.3 Reverse dependencies

Finally, if you're releasing a new version of an existing package, it's your responsibility to check that downstream dependencies (i.e. all packages that list your package in the `Depends`, `Imports`, `Suggests` or `LinkingTo` fields) continue to work. To help you do this, devtools provides `devtools::revdep_check()`. This:

1. Sets up a temporary library so it doesn't clobber any existing packages you have installed.
2. Installs all of the dependencies of the downstream dependencies.
3. Runs `R CMD check` on each package.
4. Summarises the results in a single file.

Run `use_revdep()` to set up your package with a useful template.

If any packages fail `R CMD check`, you should give package authors at least two weeks to fix the problem before you submit your package to CRAN (you can easily get all maintainer e-mail addresses with `revdep_maintainers()`). After the two weeks is up, re-run the checks, and list any remaining failures in `cran-comments.md`. Each package should be accompanied by a brief explanation that either tells CRAN that it's a false positive in `R CMD check` (e.g. you couldn't install a dependency locally) or that it's a legitimate change in the API (which the maintainer hasn't fixed yet).

Inform CRAN of your release process: "I advised all downstream packages maintainers of these problems two weeks ago". Here's an example from a recent release of dplyr:

```
Important reverse dependency check notes (full details at
https://github.com/wch/checkresults/tree/master/dplyr/r-release);

* COPASutils, freqweights, qdap, simPH: fail for various reasons. All package
  authors were informed of the upcoming release and shown R CMD check issues
  over two weeks ago.

* ggvis: You'll be receiving a submission that fixes these issues very shortly
  from Winston.

* repra, rPref: uses a deprecated function.
```

15.4 CRAN policies

As well as the automated checks provided by `R CMD check`, there are a number of CRAN policies (`http://cran.r-project.org/web/packages/policies.html`) that must be checked manually. The CRAN maintainers will typically look at this very closely on a package's first submission.

I've summarised the most common problems below:

- It's vital that the maintainer's e-mail address is stable because this is the only way that CRAN has to contact you, and if there are problems and they can't get in touch with you they will remove your package from CRAN. So make sure it's something that's likely to be around for a while, and that it's not heavily filtered.
- You must have clearly identified the copyright holders in `DESCRIPTION`: if you have included external source code, you must ensure that the license is compatible. See Section 4.4 and Section 10.6 for more details.
- You must "make all reasonable efforts" to get your package working across multiple platforms. Packages that don't work on at least two will not normally be considered.
- Do not make external changes without explicit user permission. Don't write to the file system, change options, install packages, quit R, send information over the internet, open external software, etc.
- Do not submit updates too frequently. The policy suggests a new version once every 1-2 months at most.

I recommend following the CRAN Policy Watch (`https://twitter.com/CRANPolicyWatch`) Twitter account which tweets whenever there's a policy change. You can also look at the GitHub repository that powers it: `https://github.com/eddelbuettel/crp/commits/master/texi`.

15.5 Important files

You now have a package that's ready to submit to CRAN. But before you do, there are two important files that you should update: `README.md` which describes what the package does, and `NEWS.md` which describes what's changed since the previous version. I recommend using Markdown for these files, because it's useful for them to be readable as both plain text (e.g. in emails) and HTML (e.g. on GitHub, in blog posts). I recommend using Github flavoured Markdown, `https://help.github.com/articles/GitHub-flavored-Markdown/`, for these files.

15.5.1 README.md

The goal of the `README.md` is to answer the following questions about your package:

- Why should I use it?
- How do I use it?
- How do I get it?

On GitHub, the README.md will be rendered as HTML and displayed on the repository home page.

I normally structure my README as follows:

1. A paragraph that describes the high-level purpose of the package.
2. An example that shows how to use the package to solve a simple problem.
3. Installation instructions, giving code that can be copied and pasted into R.
4. An overview that describes the main components of the package. For more complex packages, this will point to vignettes for more details.

15.5.2 README.Rmd

If you include an example in your README (a good idea!) you may want to generate it with R Markdown. The easiest way to get started is to use devtools::use_readme_rmd(). This creates a template README.Rmd and adds it to .Rbuildignore. The template looks like:

~~~
---
output:
  md_document:
    variant: markdown_github
---

<!-- README.md is generated from README.Rmd. Please edit that file -->

```{r, echo = FALSE}
knitr::opts_chunk$set(
  collapse = TRUE,
  comment = "#>",
  fig.path = "README-"
)
```
~~~

This:

- Outputs Github flavoured Markdown.
- Includes a comment in README.md to remind you to edit README.Rmd, not README.md.
- Sets up my recommended knitr options, including saving an image to README-chunkname.png (which is automatically .Rbuildignored.)

You'll need to remember to re-knit README.Rmd each time you modify it. If you use git, use_readme_rmd() automatically adds the following "pre-commit" hook:

```
#!/bin/bash
if [[ README.Rmd -nt README.md ]]; then
  echo "README.md is out of date; please re-knit README.Rmd"
  exit 1
fi
```

This prevents git commit from succeeding unless README.md is more recent than README.Rmd. If you get a false positive, you can ignore the check with git commit --no-verify. Note that git commit hooks are not stored in the repository, so every time you clone the repo, you'll need to run devtools::use_readme_rmd() to set it up again.

15.5.3 NEWS.md

The README.md is aimed at new users. The NEWS.md is aimed at existing users: it should list all the API changes in each release. There are a number of formats you can use for package news, but I recommend NEWS.md. It's not supported by CRAN (so you'll need to run devtools::use_build_ignore("NEWS.md")), but it's well supported by GitHub and is easy to re-purpose for other formats.

Organise your NEWS.md as follows:

- Use a top-level heading for each version: e.g. # mypackage 1.0. The most recent version should go at the top.

- Each change should be included in a bulleted list. If you have a lot of changes you might want to break them up using subheadings, ## Major changes, ## Bug fixes etc. I usually stick with a simple list until just before releasing the package when I'll reorganise into sections, if needed. It's hard to know in advance exactly what sections you'll need.

- If an item is related to an issue in GitHub, include the issue number in parentheses, e.g. (#10). If an item is related to a pull request, include the pull request number and the author, e.g. (#101, @hadley). Doing this makes it easy to navigate to the relevant issues on GitHub.

The main challenge with NEWS.md is getting into the habit of noting a change as you make a change.

15.6 Release

You're now ready to submit your package to CRAN. The easiest way to do this is to run `devtools::release()`. This:

- Builds the package and runs `R CMD check` one last time.
- Asks you a number of yes/no questions to verify that you followed the most common best practices.
- Allows you to add your own questions to the check process by including an unexported `release_questions()` function in your package. This should return a character vector of questions to ask. For example, httr has:

  ```
  release_questions <- function() {
    c(
      "Have you run all the OAuth demos?",
      "Is inst/cacert.pem up to date?"
    )
  }
  ```

  This is useful for reminding you to do any manual tasks that can't otherwise be automated.
- Uploads the package bundle to the CRAN submission form (`http://cran.r-project.org/submit.html`) including the comments in `cran-comments.md`.

Within the next few minutes, you'll receive an email notifying you of the submission and asking you to approve it (this confirms that the maintainer address is correct). Next the CRAN maintainers will run their checks and get back to you with the results. This normally takes around 24 hours, but occasionally can take up to 5 days.

15.6.1 On failure

If your package does not pass `R CMD check` or is in violation of CRAN policies, a CRAN maintainer will e-mail you and describe the problem(s). Failures are frustrating, and the feedback may be curt and may feel downright insulting. Arguing with CRAN maintainers will likely waste both your time and theirs. Instead:

- Breathe. A rejected CRAN package is not the end of the world. It happens to everyone. Even members of R-core have to go through the same process and CRAN is no friendlier to them. I have had numerous packages rejected by CRAN. I was banned from submitting to CRAN for two weeks because too many of my existing packages had minor problems.

- If the response gets you really riled up, take a couple of days to cool down before responding. Ignore any ad hominem attacks, and strive to respond only to technical issues.
- If a devtools problem causes a CRAN maintainer to be annoyed with you, I am deeply sorry. If you forward me the message along with your address, I'll send you a hand-written apology card.

Unless you feel extremely strongly that discussion is merited, don't respond to the e-mail. Instead:

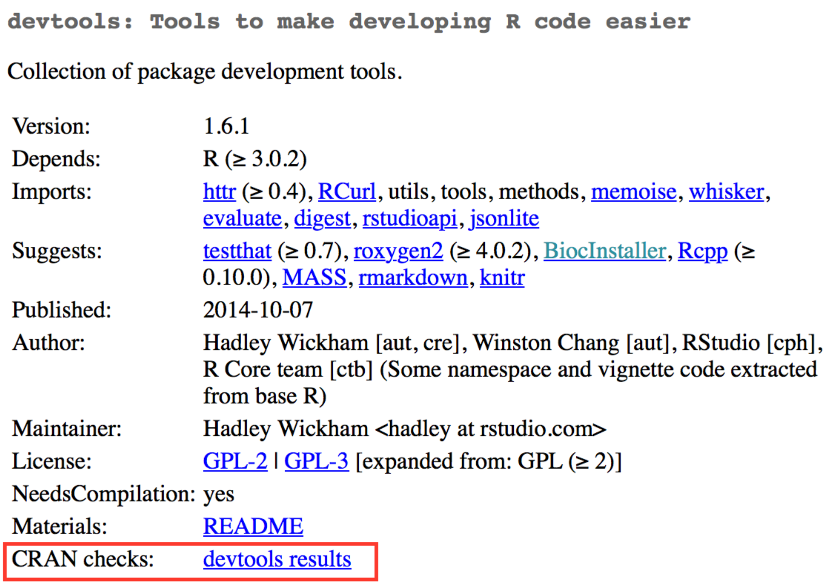
- Fix the identified problems and make recommended changes. Re-run `devtools::check()` to make sure you didn't accidentally introduce any new problems.
- Add a "Resubmission" section at the top of `cran-comments.md`. This should clearly identify that the package is a resubmission, and list the changes that you made.

```
## Resubmission
This is a resubmission. In this version I have:

* Converted the DESCRIPTION title to title case.

* More clearly identified the copyright holders in the DESCRIPTION
  and LICENSE files.
```

- If necessary, update the check results and downstream dependencies sections.
- Run `devtools::submit_cran()` to re-submit the package without working through all the `release()` questions a second time.

15.6.2 Binary builds

After the package has been accepted by CRAN it will be built for each platform. It's possible this may uncover further errors. Wait 48 hours until all the checks for all packages have been run, then go to the check results package for your package:



devtools: Tools to make developing R code easier

Collection of package development tools.

Version:	1.6.1
Depends:	R (≥ 3.0.2)
Imports:	httr (≥ 0.4), RCurl, utils, tools, methods, memoise, whisker, evaluate, digest, rstudioapi, jsonlite
Suggests:	testthat (≥ 0.7), roxygen2 (≥ 4.0.2), BiocInstaller, Rcpp (≥ 0.10.0), MASS, rmarkdown, knitr
Published:	2014-10-07
Author:	Hadley Wickham [aut, cre], Winston Chang [aut], RStudio [cph], R Core team [ctb] (Some namespace and vignette code extracted from base R)
Maintainer:	Hadley Wickham <hadley at rstudio.com>
License:	GPL-2 | GPL-3 [expanded from: GPL (≥ 2)]
NeedsCompilation:	yes
Materials:	README
CRAN checks:	devtools results

Prepare a patch release that fixes the problems and submit using the same process as above.

15.7 Prepare for next version

Once your package has been accepted by CRAN, you have a couple of technical tasks to do:

- If you use GitHub, go to the repository release page. Create a new release with tag version `v1.2.3` (i.e. "v" followed by the version of your package). Copy and paste the contents of the relevant `NEWS.md` section into the release notes.

- If you use git, but not GitHub, tag the release with `git tag -a v1.2.3`.

- Add the `.9000` suffix to the `Version` field in the DESCRIPTION to indicate that this is a development version. Create a new heading in `NEWS.md` and commit the changes.

15.8 Publicising your package

Now you're ready for the fun part: publicising your package. This is really important. No one will use your helpful new package if they don't know that it exists.

Start by writing a release announcement. This should be an R Markdown document that briefly describes what the package does (so people who haven't used it before can understand why they should care), and what's new in this version. Start with the contents of `NEWS.md`, but you'll need to modify it. The goal of `NEWS.md` is to be comprehensive; the goal of the release announcement is to highlight the most important changes. Include a link at the end of the announcement to the full release notes so people can see all the changes. Where possible, I recommend showing examples of new features: it's much easier to understand the benefit of a new feature if you can see it in action.

There are a number of places you can include the announcement:

- If you have a blog, publish it there. I now publish all package release announcements on the RStudio blog (`http://blog.rstudio.org/author/hadleywickham/`).
- If you use Twitter, tweet about it with the #rstats hashtag.
- Send it to the r-packages mailing list (`https://stat.ethz.ch/mailman/listinfo/r-packages`). Messages sent to this list are automatically forwarded to the R-help mailing list.

Chapter 16

Congratulations!

You have released your first package to CRAN and made it to the end of the book!