

# Beyond Spreadsheets

with

# R

Jonathan Carroll

# MEAP

1



MANNING





**MEAP Edition**  
**Manning Early Access Program**  
**Beyond Spreadsheets with R**  
**Version 4**

Copyright 2018 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

**Licensed to Asif Qamar <asif@asifqamar.com>**

# welcome

---

This book is designed with the non-programmer in mind. A gentle introduction to programming concepts, terminology, and structure using the R language for people interested in strengthening their data analysis skills beyond spreadsheets. Be guided through the power and the quirks of the R language and the benefits of working within a well-designed coding suite; RStudio.

Learn how to produce robust, reproducible, maintainable data processing pipelines through real examples with working code. Identify the differences between data types and how the R language interacts with them. Construct functions to perform tasks repeatedly and flexibly. Manage and interrogate your data assets and produce insightful data-driven graphics with just a few lines of code.

With a focus on the core functionality you'll want to know in order to start your journey towards becoming a data scientist, finance guru, modeller, sports analyst, manager, information visualiser, hobbyist (or any number of other applications for which R is a great choice) you'll build up knowledge of how to communicate with the computer through a casual, approachable style which doesn't assume you have a degree in software engineering. Suitable for beginners up to those already familiar with the language, there's something here for everyone.

— Dr. Jonathan Carroll

# *brief contents*

---

## *Front matter*

- 1 Introducing Data and the R Language*
- 2 Getting to Know R Data Types*
- 3 I Want To Make New Data Values*
- 4 Understanding the Tools We'll Use — Functions*
- 5 I Want To Combine Data Values*
- 6 I Want To Select Certain Data Values*
- 7 I Want To Do Something With Lots of Data*
- 8 I Want To Do Something Conditionally (Control Structures)*
- 9 I Want I Want To Visualise My Data (Plotting)*
- 10 I Want To Do More With My Data (Extensions)*

## **APPENDICES:**

- A Installing R*
- B Installing RStudio*
- C base R graphics*



## Preface

Data munging - manipulating raw data - is a cornerstone of data science. Munging techniques include cleaning, sorting, parsing, filtering, and pretty much anything else you need to make data truly useful. They say 90% of data science is preparing the data, and the other 90% is actually doing something with it. Don't underestimate how important it is to carefully prepare data; analysis interpretations hinge on getting this step right.

I love using R. It is just so very useful in so many ways. I never thought a language could be so flexible as to calculate a t-test one moment then request an Uber the next. Every word of this book has been processed by some R code; the inline results generated by actual R code and brought together using a third-party R package.

A message to those of you who have obtained a pirated copy of this book. Copyright infringement is commonly justified by those who partake in it by the notion that 'no one *loses* anything'. That's true. But it's only the infringer that *gains* anything. Many, **many** hours went into the writing and publication of this book, and without a formal sale involved, any gain you receive from reading this book goes unnoticed and unappreciated. If you have an unofficial copy of this book and have found it useful, please consider buying a legitimate copy, either for yourself or someone else you think might benefit from it.

## Acknowledgments

I would like to thank Manning Publications for the opportunity to write this book, and the overwhelmingly helpful communities on Stack Overflow and Twitter (under the [#rstats](#) hashtag).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

Licensed to Asif Qamar <[asif@asifqamar.com](mailto:asif@asifqamar.com)>

I am eternally grateful to the diverse and supportive **R** community, the majority of whom voluntarily contribute packages to improve and extend the language. The feedback, suggestions, comments, and discussions I've had regarding the contents of this book from reviewers, Twitter followers, and colleagues has helped shape this book into what it is today, and for that I thank each of them.

## **Who needs this book**

You do, of course.

Given that you're reading this, I'm guessing that you have some data (stored as a spreadsheet, perhaps) and aren't quite sure what to do with it. That's fine, great even. Maybe you want to learn something from your data. Maybe you want to find a new way to interact with it. Maybe you want to make a picture out of it. All great goals, but I'm also guessing you want to learn how to do some programming for the first time.

I'm not going to assume you know how to program already, or that you are familiar with the jargon. Perhaps you've already picked up a few programming books and been scared off by how fast they fly through the introductory material trying to get you up to speed on every nuance of the way that particular language works. Not here. We'll take things slow and work on a lot of examples together so that by the time we get to the end you'll be comfortable with doing what *you* want to do with *your* data.

I'm also not going to even mention statistics. That's a topic for someone else to cover. If you don't have a background in statistics, don't worry; it's not a requirement here. We'll be looking at R programming, not statistics (which it is very good at).

By the time you've finished reading this book, you should have a broad understanding of programming and how this is achieved with the R language; how data can be investigated, interrogated, and used to gain insights; and how to set yourself up for a robust, reproducible workflow that uses data to strengthen your conclusions.

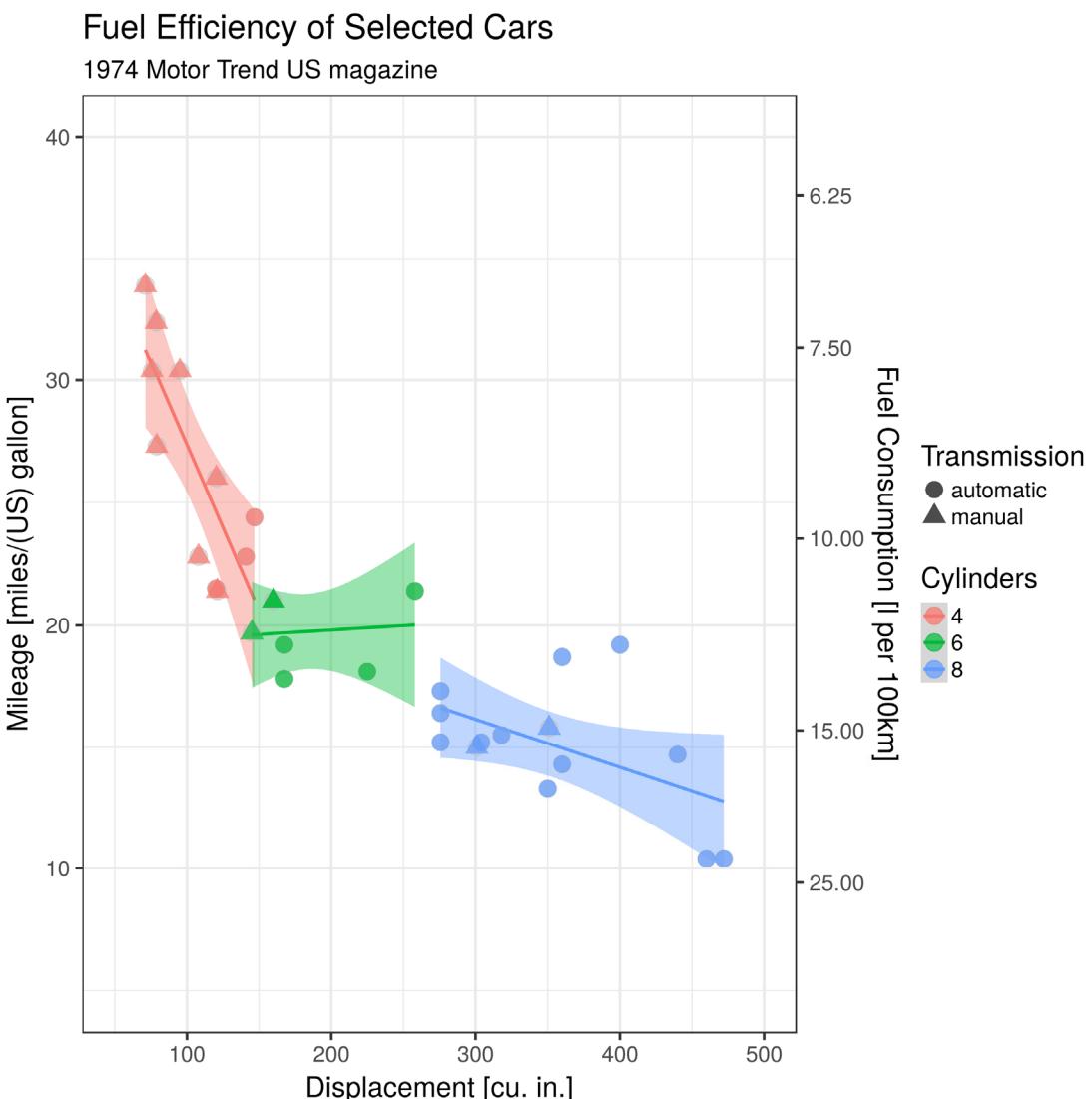
We'll see how we can take a small dataset and transform it into meaningful, publication-quality graphics with far more flexibility than any spreadsheet software can offer. With just a dozen commands, we can turn the data shown in Figure 0.1. (the `mtcars` dataset already available from within R, as shown in the RStudio data viewer)...

**Figure 0. 1. The mtcars dataset, available from within R, as viewed in the RStudio data viewer. This data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).**

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

into the graphic in Figure 0.2.

**Figure 0.2** This visualisation of the `mtcars` dataset plots the mileage (`mpg`, as well as fuel consumption in transformed units) against the engine displacement (`disp`) of the 32 vehicles, grouped both by the number of cylinders (`cyl`) and distinguished by their transmission (`am`), along with a linear fit to each cylinder group's data. This is achieved, formatting and all, in just a dozen lines of R code.



## How to read this book

I'll present each chapter to you in a no-nonsense manner; we'll cover what's important and what's likely to become an issue if you're not careful. This won't cover *every* way

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

Licensed to Asif Qamar <[asif@asifqamar.com](mailto:asif@asifqamar.com)>

to approach a problem, or necessarily the same way that other texts approach a problem, but I'll try to show you what I consider to be the best approach first, and back that up with some alternatives which you may be likely to also encounter in other reading. The goal here is to make you a competent and productive R user, which may mean showing you how to do things the slow way (as well as the fast way).

## Formatting

New terms and definitions will be presented in *italics* when they are first mentioned.

Code samples and data values will be presented in a monospace font, either inline (for mentions of code) such as `str(mtcars)` or in code blocks, for examples you should try yourself, such as

```
myData <- head(mtcars, n = 2)
```

When a code sample produces output, this will be shown below the input with the prefix `#>` and you should generally expect to see the same if you run the code yourself.<sup>1</sup> Don't worry if you try to run the lines starting with `#>`; they will be ignored by R

```
myData
#>          mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4   21   6 160 110  3.9 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag 21   6 160 110  3.9 2.875 17.02  0  1    4    4
```

Options which are available via a menu will be presented as a sequence of selections to make, such as **File ▶ Save ▶ OK** or as buttons to click such as **Cancel**. Keys you need to press will be specified such as `Enter`.

Examples with which you should follow along will be presented as blocks of annotated code, such as

### Example 0.1 Read some data from a .csv file and calculate the average height value

```
peopleData <- read.csv(file = "people.csv") ①
summary(peopleData)      ②
mean(peopleData$height) ③
```

- ① Read the data from the .csv file into a `data.frame`
- ② `summary()` acting on a `data.frame` returns a column-wise 5-number summary
- ③ We can take the `mean()` of a column of values

I will use several blocks to highlight certain features along the way:

---

<sup>1</sup> The output for the vast majority of examples has been generated by R itself in the course of generating this book.



### Sidenotes

Additional information, historical curiosities, or other notes will be presented in a block like this one.



### Important Notes

When a piece of information is particularly critical or important, it will be presented in a block like this one.



### Trip-ups

R won't always stop you from doing something you don't intend. In fact, some times it will seem to be actively trying to catch fire. Where fires are easily started, they will be explained in a block like this one to help you extinguish the flames.



### Errors

Errors produced by R will appear in a block like this one. The precise wording of the error may differ very slightly between versions.



### Alternative Approaches

There are typically many ways to solve a problem using R, and only the simplest will be discussed in any detail here. Where a better solution exists (but requires more information) it will be noted in a block like this one which will give you enough information to go find out more yourself.



### Toy Blocks (Examples)

Some examples of code I will present won't actually run, and these will be presented in a block like this one. Often this is to show you what a pattern would look like without any expectation for you to evaluate it. By all means, see if you can use these blocks to build something of your own and make it work.

Throughout the book I'll also show you what a spreadsheet equivalent starting point might look like. I will use LibreOffice, which looks like Figure 0.3 but the concepts will usually extend to Excel, Google Sheets, or whichever spreadsheet software you usually use.

**Figure 0.3. An example of cells selected in LibreOffice (Linux).**

	A	B
1	1	x
2	2	y
3	3	z
4		
-		

## Structure

As we progress through the book together, there will be lots of examples with which I hope you will follow along. Don't just read them, run them on your computer yourself and see if you get the same answers. Then try a variation on the example and see if you get the result you expect. If you get something different - that's great! It means you've found something to learn from, and your next task will be to understand why the result is what it is.

I will try to progressively build up your knowledge of the relevant programming and R-specific terms, so don't be afraid to go back and revise if something seems unfamiliar.

## Getting started

What you will need:

- This book.
- A computer.
- A desire to learn something.

Really, that's about it. R is a free (as in speech — openly available — and as in beer — it costs nothing) language, and we'll be using more free software to interact with it. You will probably need an internet connection to download the (free) software, but after that the majority of examples will work offline.

Follow along with the examples as they appear. Try different values and see if you get the result you expect. Break things, and try to understand what happened. It's very difficult to end up in a situation which can't be resolved by simply restarting R, so feel free to experiment.

This book won't necessarily direct you towards how to solve **your** specific problems, but it should give you enough of a comprehension of the language and its ecosystem for you to begin working out what tools you might need to use. If you're working in genomics there's a good chance you'll need some more advanced tools provided by the [Bioconductor project](#). Many of the concepts and structures used there extend from those you'll learn about in this book (though I won't cover those here).

## Where to Find More Help

[Stack Overflow](#) is an immensely useful source of information under the r tag, but is

frequently overrun with poorly researched questions and thankless responses. Take the time to figure out if your question already has an answer (which happens regularly given how many questions have been asked) before insisting that someone else solve your problem.

If all else fails, typing what terms you do know and `r` or `rstats` into a search engine (such as Google) tends to produce some useful results more often than not.

The R Weekly site (<https://rweekly.org>) provides a weekly summary of the most interesting R posts from around the web. R-bloggers (<https://r-bloggers.com>) provides a syndication of many popular R-related blogs and has fresh content daily. Follow along with some of these that align with your interests and you're bound to come across some useful tips.

Lastly, reach out to your local community, either in person (e.g. [Meetup](#)) or online (Twitter, `#rstats`).

## About This Book

This book was written in the Asciidoc plain-text markup language using emacs and RStudio. The R code herein was evaluated using a custom package library defined via the `switchr` R package and intertwined amongst the source using the `knitr` R package.

The session information describing the environment defining this custom library is

```
#> setting  value
#> version   R version 3.4.3 (2017-11-30)
#> system    x86_64, Linux-gnu
#> ui        X11
#> Language  en_AU:en
#> collate   en_AU.UTF-8
#> tz        Australia/Adelaide
#> date      2018-01-23
#>
#> package      * version  date      source
#> assertthat    0.2.0    2017-04-11 CRAN (R 3.4.3)
#> backports     1.1.2    2017-12-13 CRAN (R 3.4.3)
#> base         * 3.4.3    2017-12-01 Local
#> bindr         0.1      2016-11-13 CRAN (R 3.4.3)
#> bindrcpp     0.2      2017-06-17 CRAN (R 3.4.3)
#> broom         0.4.3    2017-11-20 CRAN (R 3.4.3)
#> cellranger   1.1.0    2016-07-27 CRAN (R 3.4.3)
#> cli          1.0.0    2017-11-05 CRAN (R 3.4.3)
#> colorspace   1.3-2    2016-12-14 CRAN (R 3.4.3)
#> commonmark   1.4      2017-09-01 CRAN (R 3.4.3)
#> compiler     3.4.3    2017-12-01 Local
#> crayon       1.3.4    2017-09-16 CRAN (R 3.4.3)
#> crosstalk   1.0.0    2016-12-21 CRAN (R 3.4.3)
#> curl          3.1      2017-12-12 CRAN (R 3.4.3)
#> data.table   1.10.4-3  2017-10-27 CRAN (R 3.4.3)
#> datasauRus   * 0.1.2    2017-05-08 CRAN (R 3.4.3)
#> datasets     * 3.4.3    2017-12-01 Local
```

#> devtools	*	1.13.4	2017-11-09	CRAN	(R 3.4.3)
#> digest		0.6.14	2018-01-14	CRAN	(R 3.4.3)
#> dplyr	*	0.7.4	2017-09-28	CRAN	(R 3.4.3)
#> evaluate		0.10.1	2017-06-24	CRAN	(R 3.4.3)
#>forcats	*	0.2.0	2017-01-23	CRAN	(R 3.4.3)
#> foreign		0.8-67	2016-09-13	CRAN	(R 3.3.1)
#> ggplot2	*	2.2.1	2016-12-30	CRAN	(R 3.4.3)
#> glue		1.2.0	2017-10-29	CRAN	(R 3.4.3)
#> graphics	*	3.4.3	2017-12-01	Local	
#> grDevices	*	3.4.3	2017-12-01	Local	
#> grid		3.4.3	2017-12-01	Local	
#> gtable		0.2.0	2016-02-26	CRAN	(R 3.4.3)
#> haven		1.1.1	2018-01-18	CRAN	(R 3.4.3)
#> here	*	0.1	2017-05-28	CRAN	(R 3.4.3)
#> hms		0.4.0	2017-11-23	CRAN	(R 3.4.3)
#> htmltools		0.3.6	2017-04-28	CRAN	(R 3.4.3)
#> htmlwidgets	*	1.0	2018-01-20	CRAN	(R 3.4.3)
#> httpuv		1.3.5	2017-07-04	CRAN	(R 3.4.3)
#> httr	*	1.3.1	2017-08-20	CRAN	(R 3.4.3)
#> jsonlite		1.5	2017-06-01	CRAN	(R 3.4.3)
#> knitr	*	1.18	2017-12-27	CRAN	(R 3.4.3)
#> lattice		0.20-35	2017-03-25	CRAN	(R 3.3.3)
#> lazyeval		0.2.1	2017-10-29	CRAN	(R 3.4.3)
#> Leaflet	*	1.1.0	2017-02-21	CRAN	(R 3.4.3)
#> Lubridate		1.7.1	2017-11-03	CRAN	(R 3.4.3)
#> magrittr		1.5	2014-11-22	CRAN	(R 3.4.3)
#> mapproj	*	1.2-5	2017-06-08	CRAN	(R 3.4.3)
#> maps	*	3.2.0	2017-06-08	CRAN	(R 3.4.3)
#> memoise		1.1.0	2017-04-21	CRAN	(R 3.4.3)
#> methods	*	3.4.3	2017-12-01	Local	
#> mime		0.5	2016-07-07	CRAN	(R 3.4.3)
#> misc3d		0.8-4	2013-01-25	CRAN	(R 3.4.3)
#> mnormt		1.5-5	2016-10-15	CRAN	(R 3.4.3)
#> modelr		0.1.1	2017-07-24	CRAN	(R 3.4.3)
#> munsell		0.4.3	2016-02-13	CRAN	(R 3.4.3)
#> nlme		3.1-131	2017-02-06	CRAN	(R 3.4.0)
#> openxlsx		4.0.17	2017-03-23	CRAN	(R 3.4.3)
#> parallel		3.4.3	2017-12-01	Local	
#> pillar		1.1.0	2018-01-14	CRAN	(R 3.4.3)
#> pkgconfig		2.0.1	2017-03-21	CRAN	(R 3.4.3)
#> plot3D	*	1.1.1	2017-08-28	CRAN	(R 3.4.3)
#> plyr		1.8.4	2016-06-08	CRAN	(R 3.4.3)
#> psych		1.7.8	2017-09-09	CRAN	(R 3.4.3)
#> purrr	*	0.2.4	2017-10-18	CRAN	(R 3.4.3)
#> R6		2.2.2	2017-06-17	CRAN	(R 3.4.3)
#> Rcpp		0.12.15	2018-01-20	CRAN	(R 3.4.3)
#> readr	*	1.1.1	2017-05-16	CRAN	(R 3.4.3)
#> readxl		1.0.0	2017-04-18	CRAN	(R 3.4.3)
#> reshape2	*	1.4.3	2017-12-11	CRAN	(R 3.4.3)
#> rex	*	1.1.2	2017-10-19	CRAN	(R 3.4.3)
#> rio	*	0.5.5	2017-06-18	CRAN	(R 3.4.3)
#> rlang	*	0.1.6	2017-12-21	CRAN	(R 3.4.3)
#> rmarkdown	*	1.8	2017-11-17	CRAN	(R 3.4.3)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

Licensed to Asif Qamar <[asifqamar.com](mailto:asifqamar.com)>

```
#> roxygen2      * 6.0.1    2017-02-06 CRAN (R 3.4.3)
#> rprojroot     1.3-2    2018-01-03 CRAN (R 3.4.3)
#> rstudioapi    0.7     2017-09-07 CRAN (R 3.4.3)
#> rvest          0.3.2    2016-06-17 CRAN (R 3.4.3)
#> scales         0.5.0    2017-08-24 CRAN (R 3.4.3)
#> shiny          1.0.5    2017-08-23 CRAN (R 3.4.3)
#> stats          * 3.4.3    2017-12-01 Local
#> stringi        1.1.6    2017-11-17 CRAN (R 3.4.3)
#> stringr        * 1.2.0    2017-02-18 CRAN (R 3.4.3)
#> switchr        * 0.12.6    2017-11-07 CRAN (R 3.4.1)
#> testthat       * 2.0.0    2017-12-13 CRAN (R 3.4.3)
#> tibble          * 1.4.1    2017-12-25 CRAN (R 3.4.3)
#> tidyverse       * 1.2.1    2017-11-14 CRAN (R 3.4.3)
#> tools           3.4.3    2017-12-01 Local
#> utils          * 3.4.3    2017-12-01 Local
#> withr          2.1.1    2017-12-19 CRAN (R 3.4.3)
#> xml2            1.1.1    2017-01-24 CRAN (R 3.4.3)
#> xtable          1.8-2    2016-02-05 CRAN (R 3.4.3)
```

Details for installing the specific versions of these packages are provided in [Packages Used In This Book](#).

# Introducing Data and the **R** Language



## **This chapter covers:**

- Why data analysis is important
- How to make your analysis robust
- How and why R works with data
- RStudio — your interface to R

You have your data and you want to start doing something awesome with it, right? Brilliant! I promise you, we'll get to that as soon as we can. But first, let's take a step back. Telling you to dive right in now would be like handing you a pile of different timbers, pointing you towards the workshop, and telling you to make some furniture. Instead, it's a good idea to understand both the materials and the tools we're about to use.

We'll go through what 'data' means in general — to you and those who may potentially inherit your data — because if we don't fully comprehend what we already have, then building on that won't be useful (and at worst will be flat out wrong). Poorly preparing data merely delays dealing with it properly and grows our '*technical debt*' (make it easier now, but you'll pay back that time and more later when you have difficulties working with poorly formed data).

We'll discuss how to set yourself up for a rigorous analysis (one that can be repeated) then begin working with one of the best data analysis tools available — the R programming language. For now, let's go through what it means to "have some data."

## **1.1 Data — What, Where, How?**

I said you have some data that you want to do something with, which wasn't a very precise statement. That was intentional. In fact, I can guarantee you have some data, even if you don't realise it. You might be thinking that 'data' is exclusively whatever is

stored in your Excel file, but it's so much more than that. We all have data, because it's everywhere. Before you go analysing your own data, it's important to recognise the structure of your data (both as you understand it, and as R will) and begin with a solid foundation of what it means to have some 'data'.

### **1.1.1 What is Data?**

Data exists in many forms, not just numbers and letters in a spreadsheet. That is just one of the many ways in which data might be stored. It may also be stored in a different file type such as Comma Separated Values (CSV), the words in a book, or values in a table on a webpage.



#### **Comma Separated Values**

It's common to store Comma Separated Values in a .csv file. This format is particularly useful because it is plain text; just values separated by commas. We'll return to why that's useful in [Version Control](#).

It may not be stored at all — streaming data comes as a flow of information, such as the signal your TV picks up and processes, your Twitter feed, or the output from a measuring device. We can store this data if we wish to, but often we wish to understand the flow as it's happening.

Data isn't always pretty (in fact, most times it's dirty, mundane, and seemingly uninteresting) and it isn't always in the format that we want. Having some tools at hand to manage data is a powerful advantage and critical to achieving a reliable goal, but that's only useful if you know what your data represents before you do anything further with it. "Garbage in, garbage out" warns us that we can't perform an analysis on terrible data and expect to get a meaningful result. You may very well have tried to evaluate a calculation in Excel only to have the result show up as #VALUE! because you tried to divide a number by some text, even though that 'text' looked like numbers. The types of your values (text, numbers, images, etc...) are themselves pieces of data with possible meaning behind them, and we'll learn how to best make use of this.

So what is 'good data'? What do the values you have represent?

### **1.1.2 Seeing the World as Data Sources**

We experience the world through our senses — touching, seeing, hearing, tasting, smelling, and generally absorbing life around us. Each of those input channels handle available data and our brains process them, mixing the signals together to form our picture of the world in a brilliantly complex way that we constantly take for granted.

Every time you use any of your senses you're taking a measurement of the world. How bright is the sun today? Is there a car approaching? Is something burning? Is there enough coffee left in the pot for another cup? We construct measuring tools to make life easier for us and handle some of the data consistently — thermometers to measure temperatures, scales to measure weights, rulers to measure lengths.

We go a step further and create more tools to summarise those data — car instrument

panels to simplify the internal measurements of the engine; weather stations to summarise temperature, wind, and pressure. With the digital age, we now have an overload of data sources at our disposal. The Internet provides data on virtually any and all aspects of the world we might be interested in and we create more tools again to manage these — weather, finance, social media, the number of astronauts currently in space,<sup>1</sup> lists of episodes of The Simpsons, all available at our disposal. The world is truly made up of data.

That's not to say that the data is in anyway finite. We constantly add to the available sources of data, and by asking new questions we can identify new data we wish to obtain. Data itself also generates more data; *metadata* is the additional data that describes some other data — the number of subjects in a trial, the units of a measurement, the time at which a sample was taken, the website from which the data was collected. All of these are data too, and need to be stored, maintained, and updated as they change.

You interact with data in various ways all the time, but one of the greatest achievements of the World Wide Web has been to gather, collate, and summarise our data for us in more easily digestible forms. Think for a minute about how you might have requested a taxi/cab twenty years ago, before the rise of smartphones and the 'app ecosystem'. You would: look up the phone number of a taxi company; phone them; tell the operator where you are/will be, where you want to go, what time you want to be picked up. The operator would send out the request to all drivers, one of whom would accept the request. At the end of your journey, you would pay with cash or a card transaction and receive a receipt.

Now, with the digital connections between devices, continuous Internet access, and GPS tracking, it simplifies to: opening the ride-share app; entering your destination and receiving a fare estimate, because your phone already knows where you are. The ride-share program receives this data and selects an appropriately close/available driver; exchanges your contact details in case anyone needs them; and routes the driver to you. At the end of your journey, your account is charged the appropriate amount and a receipt e-mailed to your account.

In both cases, the same data flowed between all the parties. In the latter, less people needed to be involved because the computer systems had access to the relevant data. Your phone communicated with the ride-share server, your phone communicated with the GPS system to locate itself, the ride-share server communicated with a payment server to authorise payment and the e-mail server to send the receipt.

At every point along the way, various data can be collected (anonymously where required) and saved for later analysis. How many people requested rides to the airport this month? What was the average distance travelled? What was the average wait time? Do people request more expensive trips from Apple or Android devices? Some of this was available previously, but it has never been easier to aggregate and compare.

<sup>1</sup> <http://www.howmanypeopleareinspacerightnow.com/>

Many businesses open up access to third-party developers using an *Application Programming Interface* (API) so that the data can be more systematically accessed. For example, Uber has an API that allows software to ask for fare estimates or ride histories (with authentication, to approved accounts). This is how your phone app is able to communicate with the Uber servers. Sure enough, someone has written an R package to work with this API, meaning you can include data direct from Uber in your analysis, or (in theory) request a ride direct from R.



### **Application Programming Interfaces**

Good software has a documented way to interact with it so that users and the software are able to communicate clearly and effectively. This can describe requests that can be sent to a server (and the expected responses), or just how a function should be used (and the expected return value).

### **1.1.3 Data Munging?**

Data munging refers to the cleaning up and preparation of data. Most data collected is not ready to be used in an analysis or presentation; usually there are inputs to validate, summaries to calculate, values to combine or remove, or restructuring to perform. This is a commonly overlooked aspect of using data for science but it is of vital importance. Failing to properly handle data can lead to difficulty working with it and worse, incorrect conclusions drawn from it.

The terms 'data munging', 'data wrangling', 'data science', 'data analysis', 'data hazmat', and many others are all names for more or less the same thing, with different emphasis and different trajectories depending on where the data is coming from or going to.

Most analyses (be they elaborate, sophisticated regressions or simple visualisations) begin with some form of data munging. Often that's merely reading the data into software in which case some of the handling is performed on your behalf with assumptions (these values are treated as dates, these as others as words). Having the power to control how that is performed can be essential when those assumptions are broken, or when you wish to treat your data in a particular way.

Any time you have groups of records in your data, whether that's years, patients, animals, colours, vehicles, etc... and you need to treat them differently (colour a line a certain way, only include them in an average of similar things, calculate how a quantity has changed between groups) you will perform data munging because you need to allocate records to a particular group somehow. Any other transformations, cleaning, or processing of the data also count towards data munging. It quickly becomes apparent that a large portion of any analysis can (or should) involve a lot of data munging if its conclusions are to be trusted.

### **1.1.4 What You Can Do With Well-Handled Data**

It will hopefully be clear at this point that data is potentially of great importance. It is routinely more than just numbers in a table. Medical data often represents real human lives and the effect that a particular intervention has had, be that lifesavingly positive

or tragically negative. These effects are not always immediately obvious to someone viewing them from a given perspective, so it is the role of the data analyst (professional or incidental) to extract patterns from data in order to make a decision.

Analysis of data is often useful in extracting non-obvious patterns. For example, while you might recognise a pattern to the sequence



```
#> 2 4 6 8 10 12 14 16 18 20
```

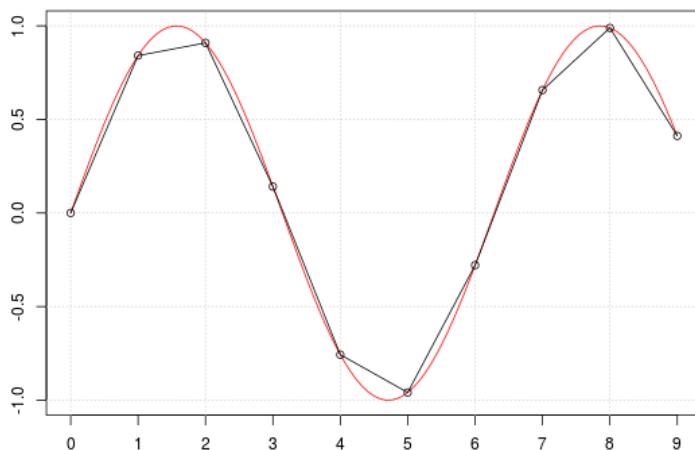
(counting by twos) it may not be so clear what the pattern is in the following data



```
#> 0.000 0.841 0.909 0.141 -0.757 -0.959 -0.279 0.657 0.989 0.412
```

until you visualise the data (which was generated with a `sin()` function) as shown in Figure 1.1.

**Figure 1.1. A pattern emerges. These points were generated with a `sin()` function at the values 0, 1, ..., 9. The smooth `sin()` function is also plotted here in red.**

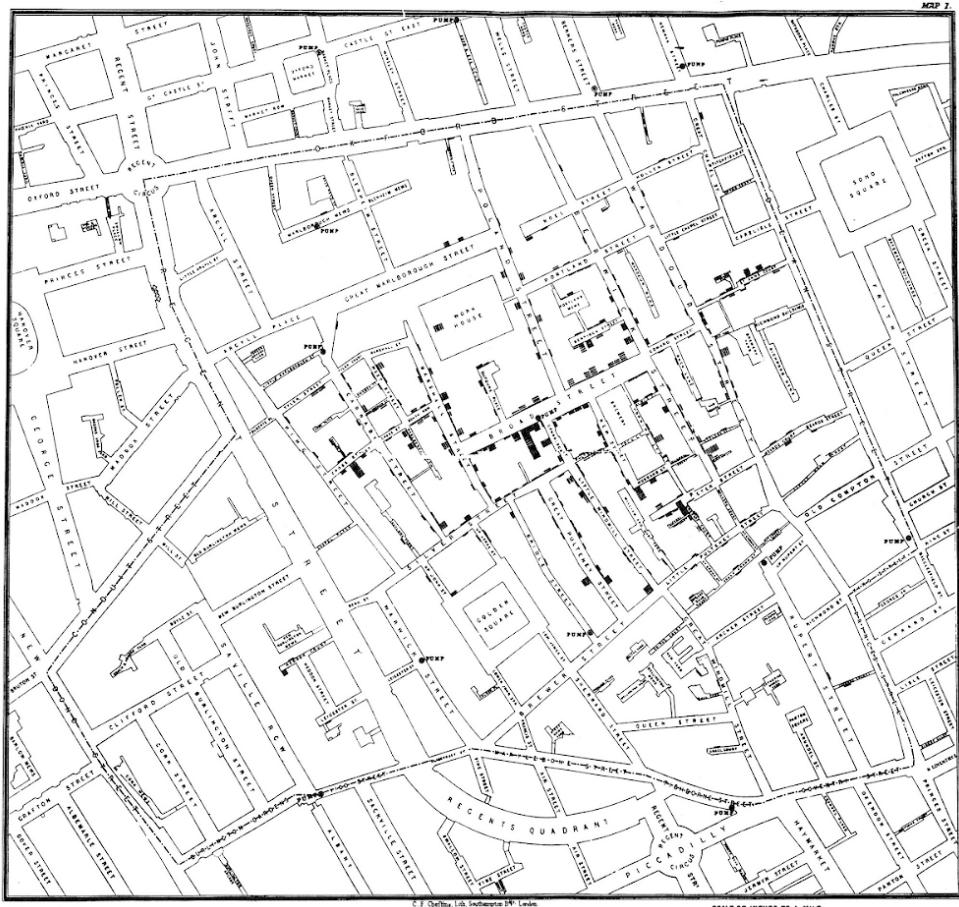


Having the right tools at hand to analyse our data means we can identify hidden patterns, forecast new information, and learn from our data.

A classic example of data analysis is that of John Snow and the 1854 Broad Street cholera outbreak in London. People were dying in the hundreds within a particular district at a time when sewerage infrastructure was all but non-existent and the understanding of infectious diseases was highly limited. By carefully examining the locations of the cholera cases, John Snow was able to infer that the common link between them appeared to be that their closest source of water was a particular water pump on Broad Street. Once the pump was disabled, cases of cholera diminished significantly. In this case, the data was in plain sight — the locations of cholera cases

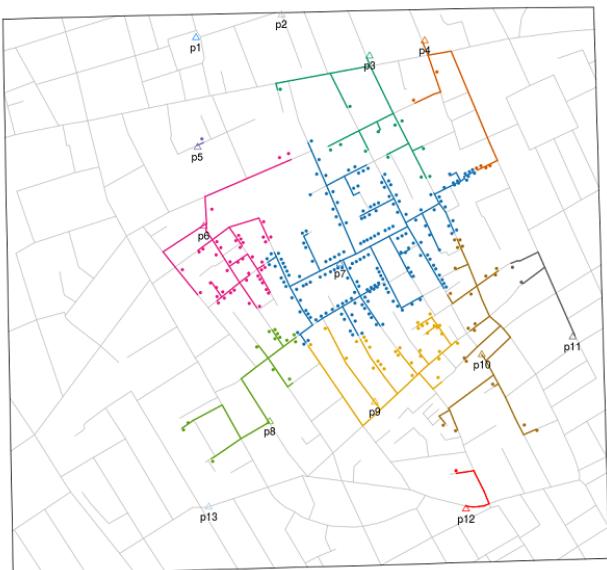
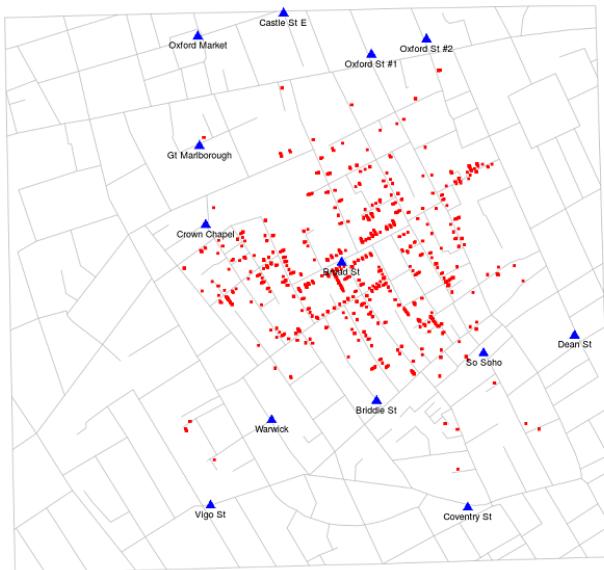
— but the pattern and connection was not immediately apparent. See Figure 1. 2.

**Figure 1.2. The Broad Street Cholera Map, by John Snow (Public domain), via Wikimedia Commons. Dots indicate pump locations, while cases of cholera are marked with stacked bars along streets.**



Perhaps unsurprisingly, there are several R packages available to interact with this data; the raw data can be found in the `HistData` package and a further graphical analysis in the `cholera` package, resulting in Figure 1. 3.

**Figure 1.3. Further analyses of the Broad Street cholera data produced using the HistData (left) and cholera (right) R packages.**



Sometimes, a spreadsheet program such as Excel or Libre Office is a sufficient tool for this purpose. Viewing some tabular numbers together, sorting them, perhaps plotting them as a bar chart; these are all easily achieved in a wide range of software applications. When we wish to interact with the data in a more structured, formal, reproducible, and rigorous manner however, we turn to a programming language. R is an excellent choice.

### 1.1.5 Data as an Asset

Data is powerful, because data is information from which we learn. We are rarely in a situation where we have no access to any data whatsoever (not just digital), but different data comes with different responsibilities.

Weather data is relied on by many to plan their day, be they fishermen figuring out how far they should venture from the relative safety of the shore, or a vigneron growing wine-making grapes planning the likelihood of overnight frost ruining their crop. Weather forecasts are not the most raw source of data, but compiled summaries produced by digesting more raw sources of measurement.

Similarly, financial analysts provide assessments of the stock markets and provide insights into the likely day-to-day movements of critical investments. These too are generated from models which ingest high-frequency measurements of the current state of the market and provide higher-level summaries which are easier to grasp and act on.

In each of these cases there are custodians of data who are relied upon; those who make available the raw measurements in a predictable and robust manner. Should these

sources of data become corrupted, either the raw or processed sources, then those further down the chain are unable to provide reliable processing of that data, and there are potentially consequences to follow. I would personally put a lot less faith in a weather report if I knew that the raw readings had been entered by hand into a spreadsheet and the forecast created by someone remembering in which order the buttons needed to be pressed, which cells needed to be copied over to another sheet, which rows needed to be selected to be included in the calculation.

The motivation behind highlighting this fact will hopefully stay with you throughout this book — we are all part of a data chain and if we do not take care with the data while it is in our possession, then all steps which follow are subject to failure in ways that will not necessarily be apparent to those who query our data. We therefore seek to produce robust, reproducible, and transparent processing of any and all data we access and release back into the wild.

While a thorough description of reproducible research requires significantly more resources to fully detail, the following guideline will serve you well for now:

- Document how, when, and from where you obtained your data
- Provide commentary on any decisions you make during your handling of the data
- Leave raw sources of data unchanged; anything you create along the way should be documented and able to be reproduced, ideally without your involvement.



#### **Maintain the integrity of data you produce**

Reproducible research is key to trusting your results, even if they don't seem to be of great significance. It may very well be only yourself looking at the results in a year, but knowing how you produced new data is just as important as what the data tells you.

Being able to trace back through the changes that a dataset has undergone is invaluable to justifying an analysis. You may end up with a plot of median income per capita for European countries, but can you tell how the scaling was performed from that? Was the data filtered for overseas income? Was the data a sample or a census? Without knowing what steps went into the analysis, the final result raises unanswerable questions.

It's critical that any analysis you perform starts with the *right* data; data that is collected in an appropriate manner and addresses the question you are asking. That question needs to be the *right* one too, otherwise you won't learn what you are hoping to.

#### ***On the right question***

***“Far better an approximate answer to the right question, which is often vague, than the exact answer to the wrong question, which can always be made precise.”***

-- John Tukey, *founding chairman of the Princeton statistics department*

With the right data and the right question in hand, how do we go about keeping track of

everything? For that, we need to be able to properly handle not just the data and the code, but *how* it changes over time.

### **1.1.6 Reproducible Research and Version Control**

Have you ever received a file with a name like

```
mydata_final_Thurs20May_phil_fixed_final_v2.xlsx
```

Not the most succinct of names, but it hints at something much worse — that multiple copies of the file are floating around, each with a different version of the data, most of which are out of date due to some corrections or updates, and with unknown changes between versions. If someone presented a graph produced from one of these files, could you be certain which version it came from? Or if presented with the most recent graph and the one that preceded it, could you tell what had changed?

The answer to this is to not rely on the filename to store the versioning information (which it is poorly suited to doing). Instead, *version control systems* (VCS) can keep track of the changes so that you (and any collaborators you are working with):

- are always up to date with the latest version of all files,
- can review the changes between versions,
- can roll-back to any previous version,

Part of this is aided greatly by using *plain-text* files (e.g. `.txt`, `.R`, `.csv`) because a version control system can literally compare the lines of two versions and show you what's changed. Using *binary* files (e.g. `.docx`, `.pdf`) makes this more difficult to extract, but doesn't make it useless.

#### **Plain-Text File**

A file which stores its contents as numbers, letters, and punctuation, and as such can be opened in a text editor. Information in a plain-text file can be read into any system and since it has no formatting, there is no ambiguity about what each symbol represents or how to read it. This doesn't preclude storage of formatting, but that too needs to be in plain-text, such as a markup language which uses tags around values, e.g. `<b>bold text</b>` or a markdown language which uses inline modifiers, e.g. `**bold**`

#### **Binary File**

A file which stores its contents in binary (zeroes and ones) to be interpreted by suitable software. This is not readable in a text editor but has the advantage that it can encode the formatting of data, including a variety of different formats such as sounds, images, or video.

We won't cover specific version control software options here, but you should find one that works for you. Some popular options are

- git (using GitHub / GitLab),
- SVN,
- Bitbucket.

Each of these has a learning curve of its own, but pays for itself the first time you need to undo a swath of changes or deletions.

Another great benefit of version control is that you can openly share (if you like) the code which describes what you have done with your data so that someone interested (possibly another data analyst, possibly just yourself six months from now) can reproduce your work because they have the inputs and the analysis steps.

Have you ever completed working on some data and became worried that perhaps you haven't saved your file, and that you might have to go through all those steps again (if you can even remember what they were)? If you can't remember what steps you performed after just completing them, how can you trust that you did them correctly? How could someone else?

By working with a *script* of commands (you could think of these as a log of exactly what you told the computer to do) you are keeping a record of the analysis steps, and someone should be able to reach the same conclusions as you did if they start with the same data.

It's not uncommon for data to require updates, and when that happens it's easy to spot the difference between people who follow reproducible research methods and those who don't. After weeks of data processing and number crunching, someone will notice that there was a typo in column 12 of the third data set and send out an updated file, 'data3\_fixedTypo.csv'.

## **The benefits of reproducible research**

### **1. The person who doesn't follow reproducible research**

- Deletes all outputs (or saves them elsewhere)
- Opens up the new data file
- Performs all of the analysis steps as best as they can remember them
- Forgets that column 4 needs special treatment.
- Doesn't understand that the final results are more different than they should be

### **2. The person who follows reproducible research**

- Changes the input data file name in their script
- Re-runs the analysis script which contains all of the required steps and their documentation
- Knows that the only thing that has changed this time is the input data update.

Many **R** packages exist for helping us work within reproducible research frameworks, and we'll cover some of the more common ones later.

With our data at the ready, our questions screaming for answers, and our intentions focused on reproducible research through version control, the only thing we still require is a way to bring all of those together to produce some results: the R programming language.

## 1.2 Introducing R

R is a statistical programming language in that it was made for the purpose of performing statistics calculations, but it has grown to be so much more through community contributions. As a general purpose language, it is flexible enough to work with almost any data you can interact with; stored or streaming, images, text, or numbers.

Like most programming languages it has a specific *syntax* (way of writing things) that may seem confusing or odd at first, but trust me, you'll get used to it soon enough. Believe it or not, R is one of the more readable languages.

R is used both professionally and recreationally by a fast-growing number of users.<sup>2</sup> Anywhere you find data, there's a good chance you'll find someone working with R. A good metric for the popularity of R is the list of professional users of RStudio (the software we'll use to interact with R) the logos of some of which are shown in Figure 1.4.

**Figure 1.4. Professional users of R.<sup>3</sup>**



Many other companies use R as part of their data processing capabilities. Some well-

<sup>2</sup> As of 2017, it was ranked sixth in the IEEE Spectrum's top ten programming languages <https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages> and eighth in the TIOBE index of popular programming languages for 2017 <https://www.tiobe.com/tiobe-index/>.

<sup>3</sup> (rstudio.com)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

known professional users and their specific uses are

### **Professional users of R<sup>4</sup>**

#### **Genentech**

use R for data munging and visualisation, and have ties to the core R developers.

#### **Facebook**

use R for their exploratory data analysis and experimental analysis.

#### **Twitter**

use R for their data visualisation and semantic clustering.

#### **The City of Chicago**

use R to build a food poisoning monitor.

#### **The New York Times**

use R for interactive features (e.g. the Dialect Quiz and Election Forecast) and data visualisation.

#### **Microsoft**

use R for XBox matchmaking.

#### **John Deere**

use R for statistical analysis (forecasting crop yields and long-term demand for farming equipment).

#### **ANZ Bank**

use R for credit-risk analysis.

R is widely used in academic research of genetics, fisheries, psychology, statistics, and linguistics, amongst many others. Amateurs have found plenty of *fun* things to do, such as solving sodoku puzzles,<sup>5</sup> and mazes,<sup>6</sup> playing chess,<sup>7</sup> and connecting to online services such as Uber.<sup>8</sup>

In this section you'll learn how R does what it does and how you'll interact with it. As with any new tool, beginning with a proper understanding of the available features can save a lot of time down the track. To fully appreciate some of the quirks of R, we need to go right back to the start.

<sup>4</sup> (revolutionanalytics.com)

<sup>5</sup> <https://dirk.shinyapps.io/sudoku-solver/>

<sup>6</sup> <https://github.com/Vessy/Rmaze>

<sup>7</sup> <http://junkst.com/rchess/>

<sup>8</sup> <http://www.exegetic.biz/blog/2016/08/uber-a-package-for-the-uber-api/>

### 1.2.1 The Origins of R

The predecessor of R was the programming language S (for statistics) developed by John Chambers and colleagues at Bell Labs. This was commercialised in 1993 through an exclusive licence as S-PLUS which was used in a wide variety of disciplines. The community saw significant growth when R was conceived as an open-source implementation of the S language, meaning that everyday users could both see the underlying structure and build on it. Nonetheless, the new language was backwards compatible with S, and much of the weirdness that remains can be attributed to this still being the case.

In February of 2000, the first stable release of R was released by Ross Ihaka and Robert Gentleman at the University of Auckland in New Zealand. The foundations of R have since been developed by a group of volunteers; the R-core developers and through proposals submitted by the general public. Development also continues in the form of externally produced add-on packages which are officially hosted on The Comprehensive R Archive Network (CRAN) of which there were roughly 12,000 by the end of 2017, as well as many more hosted informally on code-sharing sites such as GitHub.

### 1.2.2 What It Is and What It Isn't

Classifications amongst programming languages are plentiful and largely obscure. They're also constantly argued over because their definitions are complex and require a degree in computer science to fully appreciate. While R (well, S) was originally built for statistics, it can be considered as a *General Purpose Language* (GPL) in that it isn't tied to completing just one single task.

Some languages exist purely to achieve a task within some *domain* (a specific area of interest; finance, technical drawing, machine control) and these are referred to as *Domain Specific Languages* (DSL). R is much more flexible than this because you can write your code so that it achieves whichever goal you need.

#### **Domain Specific Languages**

**“ R is not a DSL. It’s a language for writing DSLs, which is something that’s altogether more powerful.”**

-- Joe Cheng *CTO of RStudio*

---

One person may have a finance data goal in mind, while another may be interested in natural language processing, while another again may be aiming to predict what decisions a customer will make next. The common link between all of these is of course data, but R is so flexible that it provides a capable mechanism to work within each of these domains.

I'm not going to sell R to you; I think it's a great language which makes many tasks simpler and has a nice way of doing things, but I won't try to tell you it's the only way to solve your specific problem. It may not even be **the** best way, but by learning a new

language we don't try to shoehorn a solution into a problem, instead we learn more about how languages work, which helps us better identify *how* a problem might be solved, even if that means another language is more suitable. Comparing programming languages is like asking which is better; apples or oranges; as usual, it depends, or maybe even doesn't. A slice of each, please.

### **What It is**

At its most basic level, R is a useful tool for interacting with data. It stores *values* (data) and *functions* (code that interacts with data) as *variables* (names for things) and complex *objects* (structures).

In technical terms, R is an *open-source, interpreted, general purpose, functional* language.

#### **open-source**

The underlying source code can be freely obtained and (if desired) modified.

#### **interpreted**

Does not require compiling into a standalone program. Some languages require the code to be built into an executable in order to run it.

#### **general purpose**

Isn't restricted to doing just one thing in a particular domain.

#### **functional**

Uses functions operating on unchanging data, rather than depending on the current state of the system and modifying data in place.

R can be thought of as a toolbelt. You can add more tools to it if you know where to hang them; you can rearrange them to make them more user-friendly if you wish; and you can work with just a few tools or many, depending on your needs. The tools in this sense are *packages* — logical groups of documented functions (code to perform operations on data) which can be called on to produce some output; a graph, more data, a signal to process, a request to a website, or just about anything.

Without packages (and by this I include the base packages and those which are installed by default) R is merely a framework with limited capabilities. The true power comes when additional packages build on this framework to create powerful statistical functions and publication-quality graphics which themselves can be extended and modified as required.

### **What It Isn't**

Having a good tool-belt doesn't automatically mean you know how to swing a hammer, or that you'll know the difference between a Philips and a Torx screw, just

the same as having R installed won't mean that all of a sudden your data analysis procedures will become clear.

R will let you do almost anything to your data, be that a wise choice or a completely unjustified one. In some cases, it will warn you that you're doing something you possibly don't want to. At other times, it will silently produce garbage and move on to the next step as if nothing is wrong. This isn't entirely the fault of R; there's many who believe that a good programming language should "*do what you say, not what you mean*" and let users decide what is right and what is wrong.

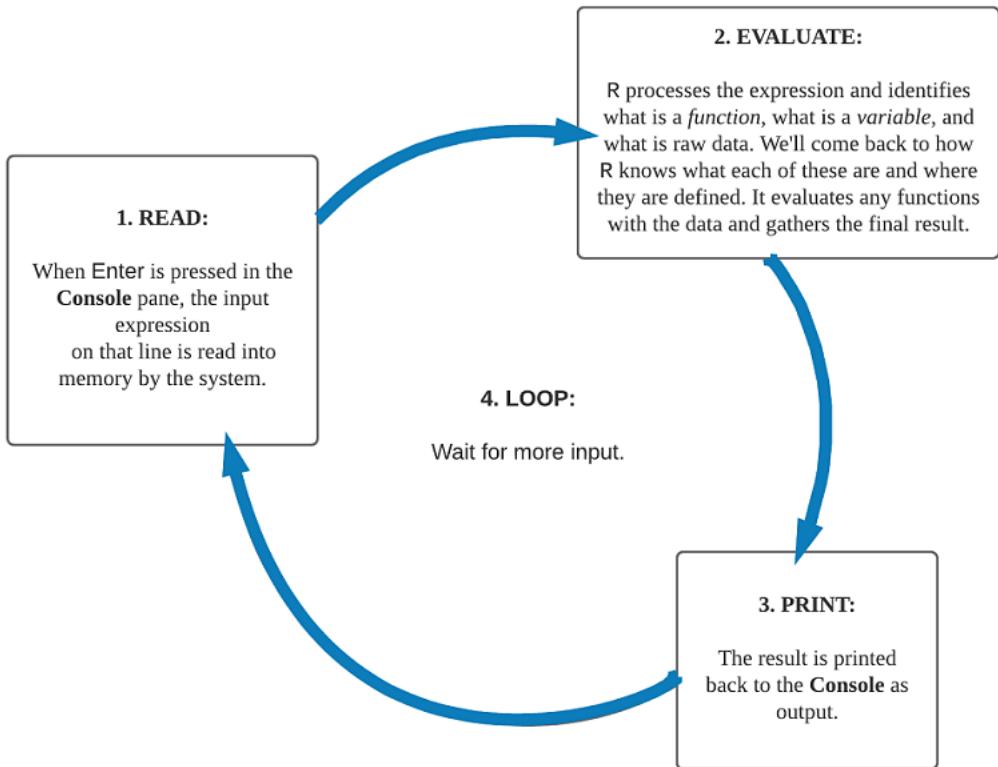
Because of the way that R works (we'll get to that shortly), it is not always the fastest method of processing data (though it's certainly not slow). Depending on your use-case, this may not be an issue at all. Sometimes the overhead of using R is an extra few minutes over some other language, with the trade-off that R code may be much more usable. Many R packages utilise R's ability to interact with other languages and strike a balance between what is processed with R and what is processed with another, more efficient language (e.g. C).

### **1.3 How R Works**

Some programming languages *compile* (build) code into an executable program. This has its advantages and disadvantages, but isn't the way that R works. Instead, R is an *interpreted* language in that the computer works with instructions one at a time (a series of these is a *script*) and the results from each instruction are presented (*returned*) to the user.

In order to operate in this way, R implements a **R**ead, **E**valuate, **P**rint, **L**oop mechanism (*REPL*), which does exactly what it sounds like. A diagram of this flow is shown in Figure 1.5. R waits patiently for your input, and once it is entered, it is **R**ead into the system, **E**valuated (calculations performed), the result is **P**rinted back to the **C**onsole (if there was any) and the entire process **L**oops back to waiting for more input.

**Figure 1.5. Read, Evaluate, Print, Loop.**



It may seem like that's a lot of capability for a language that I just said waits for input before doing anything, and that's because the *R program* ('R.exe' or the R executable that you run to *start R*) itself is actually written mainly in C which is a *compiled* language (a very memory efficient one at that). When you press **Enter**, it triggers the C code to perform the REPL operations.

Being an open-source language, the source for the code that runs *under the hood* is available for anyone to inspect. The official source for all versions back to R-0-60 is available from [svn.r-project.org/R/branches/](http://svn.r-project.org/R/branches/) which means you can see how the various components of R have changed over the years if you wish. That's just not possible with a proprietary program (closed-source) where the internal workings are only available to those working on it. With open-source software, you're even able to download the entire source, make changes to it, and compile your own personal version.<sup>9</sup>

There's also a more accessible read-only mirror hosted by Winston Chang [on Github](#) which is kept in sync with the official source hourly.

If you haven't already done so, install R on your computer. Refer to the instructions in

<sup>9</sup> The code is licensed under GPLv2 which means you can do whatever you want to it as long as you maintain the attributions of everyone who has worked on it, and don't sell it for profit or restrict access.

## Installing R.

At first you'll issue commands to R one at a time, but eventually you'll want to be able to tell R to do many things in sequence. This is called a *script* and the allusion to the lines an actor will speak is apt. An R script (typically a file ending with .R or .r) is merely a series of commands (usually one per line, but can be split over many lines) to be read in sequence by the R system and processed. This is particularly different to how a spreadsheet file behaves, where the data in its current state is preserved, but not how it got there. Some of the first lines of the script would instruct R how to prepare for the upcoming analysis, followed by how to obtain/read the raw data, then how to process it, and finally how and where to save the results. With this workflow, the analysis can be made reproducible, because armed with the raw data and the processing steps, the results can be reproduced.

R alone is sufficient to process an analysis script. This can be passed to the R processor using the command line. On Windows, depending on your exact version and installation path,<sup>10</sup> from a command line which is active in the same directory as your script file, you may be able to use



```
C:\Program Files\R\R-3.4.3\bin\R CMD BATCH yourScriptFile.R
```

On a linux or Mac system,<sup>11</sup>



```
R -f yourScriptFile.R
```

will start R and process the contents of the script file.

If you use R *interactively* (where you start R yourself and it produces a prompt, awaiting input), you can achieve this same behaviour using the source() function



```
source(file = "~/yourScriptFile.R")
```

where the *tilde* (~) in the directory name is a common placeholder for the user's home directory. Your scripts can be placed in any folder, you just need to tell R where to look.

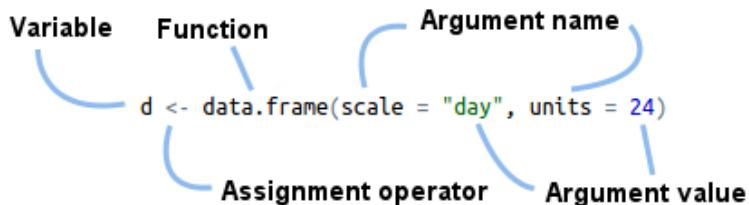
While you may be familiar with button menus in Excel, R is a command-based language. This means that we'll be telling R what to do with *expressions*; pieces of code which perform operations on our data and store the results. Let's take a moment to see what this looks like, and some of the names we'll encounter, in Figure 1. 6. Note also how different types of things are coloured (the 'syntax highlighting'); here the default colours are green for text, blue for numbers. This helps distinguish different

<sup>10</sup> Unless you set the \$PATH environment variable to search this directory.

<sup>11</sup> Assuming the installation directory is in the \$PATH.

parts of your code.

**Figure 1.6. Breakdown of some R code with terms identified; variable, assignment operator (`<-`), function, and arguments.**



Some terms used in figure 1.6. may be new to you:

#### **Variable**

A name to refer to a piece of data.

#### **Assignment operator**

Function which stores a value in a *variable*.

#### **Function**

Some code that interacts with data, *called* (invoked) with an opening ( and a closing ) (parentheses), possibly with *arguments*.

#### **Arguments**

Options passed to *functions*, separated by commas, possibly as pairs of argument names and argument values linked by an equals sign (=), e.g. `save = TRUE`.

Working with R in this way is certainly possible, but to really get a helping hand along the way, we turn to an additional piece of software that wraps around the R system and provides additional functionality.

## **1.4 Introducing RStudio**

Data is stored on your computer (or somewhere your computer can connect to) but however you interact with it you require some software to read the data, interpret what you want done to it, and to write it some sort of output or storage (either as values, an image, a sound, or something entirely different).

This can take a wide range of forms; from viewing the raw, locally stored data in a text editor such as Notepad or emacs; to displaying formatted data in a spreadsheet or database program such as Excel, Access, or Google Sheets; to viewing either unencoded or translated JSON data as it passes over the Internet with a browser such as Google Chrome or Internet Explorer; to using programming software to retrieve and manipulate data.

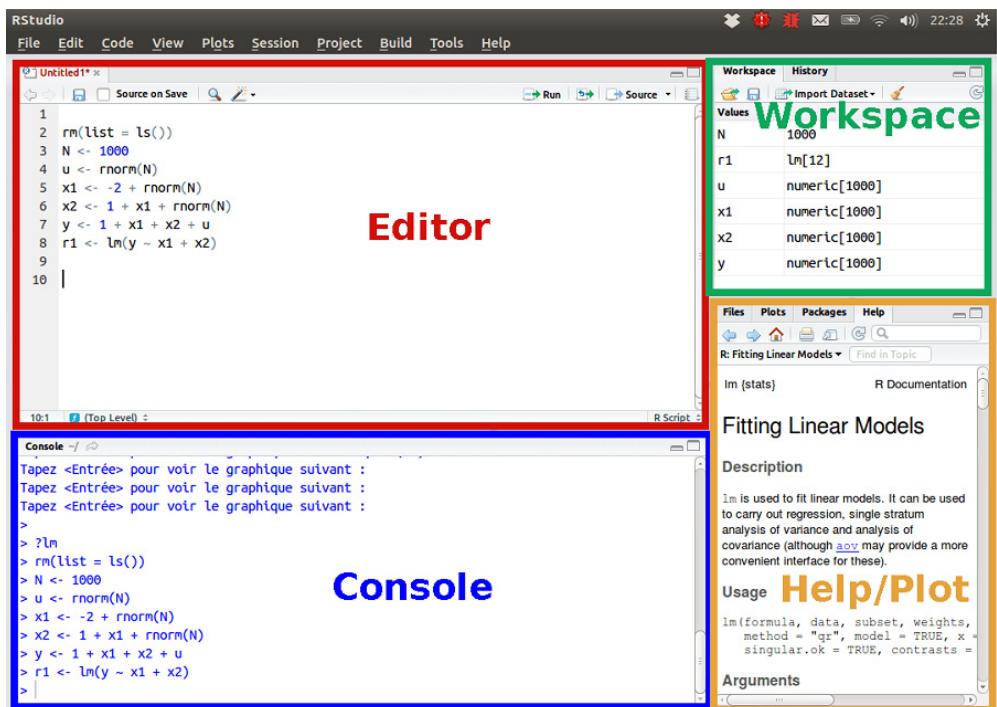
All of these have different abilities in terms of displaying and interacting with data. When it comes to using R for interacting with data, a highly sophisticated and powerful *Interactive Development Environment* (IDE) brings all of these abilities together in the form of RStudio. Here we will be able to view our data in many forms, interact with it, manipulate it, and then store it or distribute it. This IDE features an R-aware text editor for reading/writing our scripts, a Console for entering R commands, and best of all, a way to inspect the current state of the Workspace and all the defined variables.

If you haven't already done so, install RStudio on your computer. Refer to the instructions in [Installing RStudio](#).

### 1.4.1 Working with R within RStudio

RStudio divides the window into separate *panes* or sections (see Figure 1.7.). The borders of these can be dragged to expand or contract individual panes, and be arranged as you prefer via Tools → Global Options → Pane Layout. They can also be detatched from the main window to be made fullscreen.

**Figure 1.7. RStudio panes as they appear in Ubuntu/Linux (adapted from screenshot originally by PAC2 [AGPL](#), via Wikimedia Commons).**



The four panes as they appear by default are:

### *Editor*

This is where your *scripts* are written; a series of commands to be executed in order. When you first open RStudio this will be empty. Use File → New File → R Script to start a new file/script.

### *Console*

The R prompt as it would appear in a terminal. This is where you enter commands line-by-line, followed by pressing `Enter`. Results returned from R are presented here.

### *Workspace*

The values that R knows about; your data, or the *variables* you have defined appear in the Environment tab, while the history of which commands you have executed appear in the History tab.

### *Help / Plot*

Depending on which tab you have selected (e.g. the Help or Plot tab) will either display the documentation for a function or dataset, or the most recent plot produced. Also contains the Packages and Files tabs for listing installed packages and files from your computer, respectively.

There are easy ways to switch between these panes; `Ctrl/Cmd + 1` moves the cursor to the Editor pane for writing *scripts*, while `Ctrl/Cmd + 2` moves the cursor to the Console pane for interactive commands. While the cursor is on a function, `F1` will bring up the Help menu for that function. There are many other keyboard shortcuts available; try `Alt/Option + Shift + K` to bring up an extensive cheatsheet.



### Alternatives to RStudio

Of course, RStudio isn't the *only* way to use R, but I certainly find it to be the most convenient. If working with a command-line interface is more your style (and you can forego the added benefits that RStudio offers) then Rworks fine within a terminal. R can also be hooked into emacs using the 'Emacs Speaks Statistics' (ESS) emacs package. When you first install R under Windows, you will also find the RGui is installed which is a simple graphical interface.

Several alternative graphical interfaces to R are also widely used, such as R Commander or Deducer. For consistency (and because I genuinely believe it to be superior) I will present the remainder of this book as if we are working within RStudio.

RStudio lends a very helpful hand while working with R, but you can certainly do everything you need to in a terminal alone. Your textual interaction with R in that case would still match what will appear in the Console pane of RStudio, which we'll focus on now.

RStudio works nicely with git and SVN right out of the box, and I recommend you

read up on that from RStudio directly.<sup>12</sup>

Each time you start an *R session* (running the *R* program and working with the *R* language) either within RStudio or standalone, the *workspace* begins empty.<sup>13</sup> If you have the option enabled from the settings (on by default) then the files you had open last time you used RStudio will still appear in the *Editor* pane. The *Workspace* pane will not have any objects listed, and the *Console* will greet you with the welcome message

### **Listing 1.1. The R welcome message**

```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"      ①
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)                  ②

R is free software and comes with ABSOLUTELY NO WARRANTY. ③
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.                                     ④

>                                                       ⑤
```

- ① The *version* is listed here (and the date it was released) — try to keep your version as up-to-date as possible as bugfixes and improvements are regularly added. Note that this may require you to update your *package* library also.
- ② Here the *platform* is listed; typically Windows, Linux, or Mac.
- ③ As is common for free software, you get what you pay for. This is not unreasonable, and merely notes that, whilst great care is taken to ensure things work as expected, there is no formal guarantee of this as such.
- ④ Often overlooked, but these are great tips.
- ⑤ The *prompt* symbol > indicates that R is ready and waiting for our input.



#### **Kite-Eating Tree**

Since late 2011, R versions have been designated not only a version number but also a quaint name, beginning with "Great Pumpkin". The names are entirely the whim of a core developer, and while there is no strict structure, they are typically seasonally thematic and all are linked to the comic "Peanuts" by Charles M. Schulz, such as the current name "Kite-Eating Tree"

<sup>12</sup> <https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects>

<sup>13</sup> Unless you specifically set the option to start where you left off by loading the workspace image.

referencing the panel in Figure 1.8.

Figure 1.8. Peanuts, © Charles M. Schulz.



When the prompt (`>`) is visible, R is awaiting your next command. Commands can be entered either directly into the Console (better for short commands used once) which can be executed by pressing `Enter`; or built up in the Editor as a script (better for longer analyses and saving your steps) which can be executed one at a time by pressing `Ctrl + Enter` (or `Cmd + Enter` on a Mac) while the cursor is on the relevant line, or many at a time with a highlighted region.



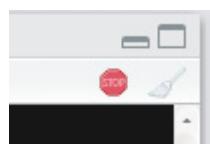
If you need to enter more than one command on a single line, you can separate these using a semi-colon (`;`), such as a `a <- 2; b <- 3`. If you begin writing a long command and are part-way through; say, have an opening (without a closing `)`) or only half of a calculation such as `2 +` and press `Enter`, R will assume you mean to insert a line-break in the command and will replace the prompt with a `+` to indicate that it's waiting for the rest of the command, which you can enter as normal. R will join all of the input lines together before executing them. In the Editor pane, commands can be split in the same way (over several lines) and executed either as a whole or in part.



Don't worry if you get stuck in this mode. Pressing `Enter` again will only insert more line-breaks. To exit this mode (or cancel entering a command at any time) press `Esc` to clear the current input and return you to the prompt (`>`).

If you're performing a complex calculation, R will indicate that it's busy by not showing that prompt until it's ready for more commands. Whatever you type into the Console will still be processed once R is available again, but you shouldn't rely on the currently running evaluation being successful. Additionally, RStudio will show a small stop sign above the Console while long-running calculations are keeping R busy, similar to the one shown in Figure 1.9.

Figure 1.9. Look for this symbol above the Console while R is running.



©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

Licensed to Asif Qamar <[asif@asifqamar.com](mailto:asif@asifqamar.com)>

Each time you start up RStudio (or start R from the command line) you are working in a *session*. Within a session the data (and functions) loaded into memory (the *workspace*) are available to be queried and modified. This is a very different mindset to adjust to if you are coming from a spreadsheet-based environment where the data is ready and waiting when you open the program back up. Here, we are saving the *raw* data and steps to produce the output, which means your work is reproducible. Nothing is permanent until it is saved.



**Packages and data that have been loaded are available only within a given session, so if you manually load a package in one session (for example using the `library()` function, which we'll cover fully in Chapter 4), read-in some data, or create anything, you'll need to repeat those steps in any other sessions for which you wish to use that package. By saving your script (a log of the commands you've entered) you can easily restart your session and catch back up to where you were. Scripts are merely plain-text commands saved to disk, and exist like any other file, independent of R sessions.**

Commands which you want R to run when it starts a session should be placed in a `.Rprofile` file in your home directory.<sup>14</sup> These could be calling `library()` on some packages you always intend to use, printing a message, or performing some common task.

If you try to exit R (either using the `q()` function or by closing the RStudio window) you will likely be prompted to "save your workspace image". This allows you to keep everything you have defined (*assigned*, which we'll cover in the next chapter) within your workspace in the session to re-load later and resume where you left off. This may or may not be something you want to do, depending on what it is you're doing, but you should try to write your code in a way that will produce the same answers if you do have to restart without saving. This helps ensure your workflow is reproducible.

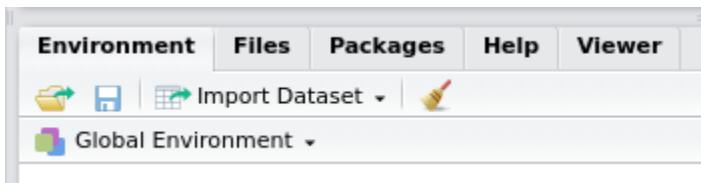
The setting can be changed if it becomes bothersome, using Tools → Global Options... → General → Save workspace to .RData on exit. I recommend you set this to not saving by default so that you always start with a fresh session.

When such a times comes when you want or need to just start from a fresh session with nothing defined and no extra packages loaded, you can of course close down R or RStudio and re-open it, but if you are using RStudio then a faster option is to press `Ctrl + Shift + F10` which will instruct RStudio to re-start the session. In some versions this will also remove all of the defined objects from the Environment. If it doesn't, you can click the broom icon to clear the Workspace, as shown in Figure 1.10.

---

<sup>14</sup> This location depends on your operating system, but can be located from R itself as `Sys.getenv('HOME')`.

**Figure 1.10. Clean up after yourself.**



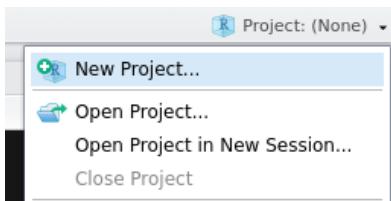
It's a good idea to do this occasionally to make sure that your script is reproducible — it's all too easy to shuffle a few lines around and convince yourself that you still get the outcome you expect when the objects already exist, but it breaks the 'sequential' nature of the script and falls apart when an object is used before it is defined.



### RStudio Projects

You'll likely work on several different things as you discover how to work with R, sometimes more than one thing at a time. RStudio makes this easy to keep track of with 'Projects'. When you create an RStudio Project, RStudio keeps track of which files you have open, where they are located, and even the positioning of the different panes, specific to that Project. Each Project has its own R session, so you can run several independently for different contexts without worrying that your finance analysis will interfere with your mapping visualisations.

I highly recommend you start each distinct context of work in a new Project. You can select Projects from the top right menu in RStudio



or create a new Project with File → New Project, or open an existing Project with File → Open Project.

RStudio will try to warn you when you do accidentally create code which doesn't make sense, possibly because it's in the wrong order, but might not if a variable is actually defined at the current state in the session. A session isn't much use until you can do something within it, so let's move on to what we can actually tell R to do with some data.

## 1.4.2 Built-in Packages (Data and Functions)

R comes pre-installed with a variety of packages that are considered as the base version of R. It also bundles in a series of useful packages that will help you do most of what you need to do. These are

```
#> [1] "base"      "compiler"   "datasets"   "graphics"   "grDevices"
#> [6] "grid"       "methods"    "parallel"   "splines"    "stats"
#> [11] "stats4"    "tcltk"     "tools"     "utils"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

Licensed to Asif Qamar <[asif@asifqamar.com](mailto:asif@asifqamar.com)>

we'll see more about these, and the functions they contain, later. For the `base` package, you can view the list of functions this provides by typing the following into the Console and pressing **Enter**

```
help(package = "base")
```

which lists these (450+ of them in R 3.4.3) alphabetically. The `stats` package provides an additional 300+ functions. You certainly won't need *all* of these, but it should be clear that there's a lot of functionality to utilise in the standard R installation, even before we reach out and install additional packages.

You don't need to do anything special to use the functions that these `base` packages provide; they're locked and loaded from the time you start up R.

In addition to useful functions, R comes pre-installed with example datasets which are useful for demonstrations. The most commonly used ones are:

#### `mtcars`

Motor Trend Car Road Tests; This data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973—74 models). This is the dataset we'll use in our examples most often. It's cliche, but cliches only get old if you've seen them too often, which in this case you probably haven't. If you need to refer to it quickly without an R session handy, a screencapture of the entire dataset is shown in figure 0.1 in the frontmatter.

#### `iris`

Edgar Anderson's Iris Data; This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris setosa, versicolor, and virginica.

#### `USArrests`

Violent Crime Rates by US State; This data set contains statistics, in arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. Also given is the percent of the population living in urban areas.

If you have read other guides to using R, you will likely have seen these mentioned. We'll use these for our examples also, but don't be afraid to test out your new skills using these and others. More example datasets (87 listed in version 3.4.3) are named and briefly described in the help menu for the package `datasets`, which you can view by entering the following into the Console

```
help(package = "datasets")
```

## 1.5 In-built Documentation

When correctly constructed, R packages, functions, and data come with some helpful

documentation. This can be accessed from within an R session with the following syntax (a question mark followed by the name of the package, function, or dataset to query), for example to learn more about the `mean()` function you would enter into the `Console`

```
?mean
```

If the name following `?` is a package, function, or dataset that R is aware of then the documentation for that will appear in the `Help` pane, otherwise an error message will appear in the console, such as

```
?nonExistentFunction
```

To search for some text in the documentation (look for something without knowing the name of it) use double question marks

```
??mean
```

This search has some limitations to say the least. Some better solutions are in progress, such as DataCamp's [RDocumentation](#), but for now most problems are covered by reading the manual (the `?documentation`) and/or searching the web.

RStudio will also help you write your R code by providing pop-up tips on syntax (hover over a function name to see the template and default arguments/options); autocomplete for known functions and arguments (pause during typing the name of a function or data value to see potential completions, select these with the and buttons, and press to see potential completions); and warnings when you've potentially misspelled a function or data value name. These options are configurable via Tools → Global Options → Code.

### 1.5.1 Vignettes

A vastly under-utilised (but invaluable when used correctly) feature of R is the ability to create an in-depth guide along with any package, known as a *vignette*. These can cover any topic, but generally highlight the usage of a few important functions that a package provides, similar to a research paper or tutorial, and at multiple page-length are much more insightful than the limited information provided in the help page, which is more of a brief usage guide.

These *vignettes* are built when you install a package, e.g.

```
# The plot3D package produces 3D plots.
# Install it using
install.packages(pkgs = "plot3D")
```

and are available from the `Console` via the `vignette()` command, with the topic (name) of a *vignette* (and potentially the package it arises from with the argument `package`)

```
# Vignette: Fifty ways to draw a volcano using package plot3D
vignette(topic = "volcano")
```

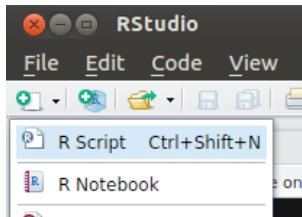
You can view all of the currently available vignettes with the command

```
browseVignettes(all = TRUE)
```

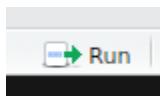
which will load a locally-hosted web page with links to each of the available *vignettes*, arranged by the package they correspond to. These are generated as either PDF files, HTML (web) files, source files for these, or raw R source chunks. If you want to learn more about a package you've installed, this is a great place to check.

## 1.6 Try It Yourself

 If you haven't already, go ahead and open up RStudio and familiarise yourself with the program. Create a new folder in your home directory where you will save your work and open a new R script



Enter some commands into the script (try adding some numbers) and see them appear in the **Console** when you evaluate them (**Ctrl** + **Enter** or **Cmd** + **Enter** or the 'Run' button just above the **Script** pane)



Take note of how commands, results, errors, and warnings appear; how lines are evaluated; and how RStudio displays different pieces of code. Don't forget to save your script regularly!

## 1.7 Summary

In this chapter you've hopefully gained a greater respect for data in terms of reproducibility, management, and integrity. You've been introduced to R as a highly powerful tool for maintaining these qualities and producing robust insights into our data.

You've learned that:

- Data is everywhere
- Handling of data comes with responsibility
- Correctly structuring data makes it more accessible
- Different types of data require different treatments
- Visualising data can uncover hidden patterns
- Reproducible research means you can defend your results

- R is used in many fields and has a long history
- R interprets your commands rather than compiles to an executable
- RStudio makes working with R code much smoother
- R is extended by many packages

New terms you've learned:

***reproducible research***

creating a reproducible body of work by storing raw data and processing steps required to achieve a result, rather than merely storing the result.

***script***

a series of commands saved in a file (usually ending in `.R` or `.r`).

***value***

a piece of data.

***variable***

a name to refer to a piece of data.

***function***

some code that interacts with data.

***object***

a complex (or simple) structure that defines a variable or function.

***package***

a collection of functions (and potentially data) that can be installed and used, which extends the functionality of R.

***session***

each running instance of R represents a *session*. Assigned variables and loaded packages/functions are only available within a session, and must be recreated if a new session is started.

***vignette***

long-form guides to using a package and its functions, stored along with installed packages.

Things to remember:

- A visible prompt (`>`) in the `Console` means R is ready for your commands.
- You can write your commands in a *script* or one at a time in the `Console`.
- To exit the session, either close RStudio via the close button, or enter the command `q()`.

- To find out more about a function, type the function name preceded by a question mark, such as `?mean`.

Now it's time to learn how R will achieve this by understanding how R interacts with data and how we can tell R what our data represents.

# Getting to Know **R** Data Types



## **This chapter covers:**

- types of data you will likely encounter
- How **R** stores data using types
- How to assign values to variables

In any programming language, data are stored by the computer in memory using *binary*.

### **Binary**

A *binary* value has only two possible states, and we can write these as TRUE and FALSE, or more commonly, 1s and 0s — in computer memory, these represent the presence or absence of an electrical charge, so ON and OFF.

We may see a 42 on the screen, but this is thanks to the computer translating what it has stored into what we expect to see. This can be extremely useful (rather than seeing 00000000 00000000 00000000 00101010) but also somewhat dangerous, as we'll later show.

There are many different ways to store a piece of data in computer memory, and by instructing the computer to treat our data as a certain type, we can change how our data is manipulated.

## **2.1 Types of Data**

**R** is what's known as a *weakly typed* language, in that the *type* of data is guessed or assumed rather than *declared* (or enforced). This can be very useful in terms of getting started because we don't need to worry about *setting* the data type whenever we're using data, but we'll also learn how to *change* the type later on if we aren't getting what we want by default.



### Strongly typed languages

In contrast, some languages such as C are *strongly typed* and in these languages the data type must be explicitly declared before it's used. This too has advantages such as knowing that the data type can't be accidentally changed.

Several data types are available in the R language and we'll go over the most common ones that you're likely to encounter. A quick summary of these is shown in Table 2.1.

**Table 2.1. Data types**

Data Type	Examples
Numbers	<ul style="list-style-type: none"> <li>• 1</li> <li>• 3.14</li> <li>• -99</li> </ul>
Text (Strings)	<ul style="list-style-type: none"> <li>• "abc"</li> <li>• "x"</li> <li>• 'I said "'Hello'"'</li> </ul>
Categories (Factors)	<ul style="list-style-type: none"> <li>• months ("Jan", "Feb", "Mar")</li> <li>• colours ("red", "blue", "green")</li> <li>• countries ("Australia", "Japan", "USA")</li> </ul>
Dates and Times	<ul style="list-style-type: none"> <li>• 2016-10-31</li> <li>• 2000-01-01 00:00::01 UTC</li> </ul>
Logicals	<ul style="list-style-type: none"> <li>• TRUE</li> <li>• FALSE</li> </ul>
Missing Values	<ul style="list-style-type: none"> <li>• NA</li> <li>• NaN</li> </ul>

#### 2.1.1 Numbers

The most common type of data is numbers in one form or another. Counts of birds, heights of trees, visitors to a website, temperatures, speeds, prices. All of these are numbers, but not all of these should be represented the same way. For some of these, it only makes sense to talk about whole numbers. The formal name for this type of number is *integer*. These are stored by the computer in a specific way. For more general numbers, where we might have a whole part and a fractional part, these are sometimes called *real* numbers. In R these are simply called *numeric* values.<sup>15</sup>

#### Integer

Meaning *whole* or *untouched*, these typically arise when we're dealing with counting things that can't be divided into smaller parts; 5 sheep, 3,000,000 people, 35 words. These have no *fractional* part to them.

<sup>15</sup> For those more interested in computer science, these are double-precision values.

**Real**

Also called *numeric* or *double*, these may have both an *integer* part and a *fractional* part, though the fractional part may be zero, in which case the value is equivalent to an integer. These arise from measurements that can take any value; 3.5 hours, 154.2 cm, 1.44 Mb.

When you type something into the R Console and press `Enter`, R evaluates the expression and returns the value.

Try it out by entering 3.2 into the Console and pressing `Enter`. You should see

```
3.2
#> [1] 3.2
```

Don't worry about the [1] for now, we'll come back to that. You may very well ask "*What type did I just use?*" We didn't specify, so R took a reasonable guess. We can examine the storage type of something (as well as the general structure) using the built-in command `str()` (short for structure).

Try entering the following into the Console:

```
str(object = 3.2)
#> num 3.2
```

Here we see that R reports that the *object* 3.2 is of type `num` (short for `numeric`) and has the value 3.2. Let's try that again with an *integer*:

```
str(object = 7)
#> num 7
```

R reports that this is object is still a *numeric* type. Maybe we weren't specific enough?

```
str(object = 7.0)
#> num 7
```

Still *numeric*. So, what happened? By default, R will assume that a number entered with or without a decimal is of type *numeric*. If you do want to specify that you explicitly want an *integer* value, you can do so by typing an L after the value:<sup>16</sup>

```
str(object = 7L)
#> int 7
```

In each of these cases, the computer takes the value we give it (3.2 or 7L) and translates that into something it can store (an *object*), which we can represent by 1s and 0s. There are various schemes for doing this, and they are typically different for *numeric* and *integer* types, but for now it is sufficient to know that those two types are different.

The `str()` function is very useful for inspecting the *structure* of an object, but when all we are interested in is the *type* of an object, a simpler function can help — the `typeof()` function. Let's try that with a value

<sup>16</sup> Why L? Well, I looks too much like a 1, but the true reasoning behind the choice may have more to do with 'long' numbers in the C language. 'Literal' is a useful mnemonic for this.

```
typeof(x = 7)
#> [1] "double"
```

The `typeof()` function reports slightly differently to `str()`, but this means the same — *numeric*, *real*, or *double* all mean the same thing.<sup>17</sup>



Note that the *argument names* used in these two examples are different; `str()` has the *argument* object while `typeof()` has the argument `x`. Every function has its own definitions of what its arguments are named, and there's not always rhyme or reason to these. Trying to use the wrong argument name leads to trouble

```
typeof(object = 7)
```



Error: unused argument (object = 7)

Alternatively, trying with an explicit integer gives

```
typeof(x = 7L)
#> [1] "integer"
```

Part of what makes this so confusing is that when R prints a value *equivalent* to an integer (i.e. `7L`, `7`, or `7.0`) it just prints the value without a decimal (i.e. `7`) so all of these look the same on the screen

```
7
#> [1] 7

7.0
#> [1] 7

7L
#> [1] 7
```

One important thing to keep in mind is that R (or almost any programming language) won't keep track of your measurement *units* for you, it will only store the number *values*.

## Units

Measurements are composed of two parts: a value and a scale. We may tend to abbreviate this when we talk ("Driving at 60") but this only makes sense if we assume we are all familiar with the *units* (kilometers per hour or miles per hour?). You may store population counts on the scale of 'millions of people', but the *value* you use may be only `318.9`.

A helpful way to ensure that your calculations are consistent with respect to units is to name your variable according to the units it refers to (we'll cover this shortly).



## On keeping track of units

In September of 1999 the Mars Climate Orbiter began its descent into the Martian

---

<sup>17</sup> If you do a lot of high-level mathematics you may also require the *complex* type such as `3 + 2i`, which `str()` and `typeof()` report as `cplx` and `complex`, respectively.

atmosphere. The optimal altitude for the required maneuvers was 226 km (140 miles), but calculations performed later showed it was actually on a path that would place it a mere 57 km (35 miles) from the surface. The probe was destroyed by the atmospheric stress. The team at Lockheed Martin made their calculations for the thrusters in pound-seconds, while the NASA trajectory calculating software expected Newton-seconds. Units are important!

There are other ways to input numbers. If we need to input very large numbers such as millions, we can use something similar to *scientific notation*<sup>18</sup> where we specify an *exponent* in the power of tens with an e or E. This means that entering 'five million' can be entered simply as

```
5e6 (1) 1
#> [1] 5e+06
```

① equivalently 5E6.

since 1,000,000 has 'six zeroes'. The part of the number in front of e or E can have decimal places if needed, so 'one million, two hundred thousand' (i.e. 'one point two million', 1,200,000) can be entered as

```
1.2e6
#> [1] 1200000
```

If we are more specific about the decimal places than the power of ten exponent, this will carry through

```
3.14159e3
#> [1] 3141.59
```

If we want to ensure that this value is actually an *integer*, we can add the Lsuffix to the *end* of the input

```
typeof(x = 1.2e6L)
#> [1] "integer"
```

and R will make the *entire* value an *integer*. Note that there is no way to use a *numeric* value in the power of ten exponent; it is always assumed to be an integer, and entering something with a decimal generates an error.

While it may be tempting to enter large numbers with thousands-separating commas or periods (e.g. 1,200,000) R won't know what to do with this

```
1,200,000
```

 Error: <text>:1:2: unexpected ',' #> 1: 1, #> ^

This is particularly important when reading in numeric data from an external real-world source. We'll see how we can deal with this situation in ["Text Substitutions"](#).

<sup>18</sup> Refer to [https://en.wikipedia.org/wiki/Scientific\\_notation](https://en.wikipedia.org/wiki/Scientific_notation)

## 2.1.2 Text (*Strings*)

Numbers are the most common data type, but the next most common would be *strings*. These are groups of *characters*, *letters*, or any form of *text*. R treats these differently because they **are** different. They represent an entirely different type of data, but a much richer one.

Strings are entered into R using either double quotes ("apple") or single quotes ('apple') and both are treated the same. We can examine the structure of this data

```
str(object = "apple")
#> chr "apple"
```

R reports that the object "apple" is of type *char* (short for *character*) and this is how strings, composed of zero or more characters, are stored. R doesn't particularly mind what's inside the quotes, so for example, an empty string is perfectly fine

```
str(object = "")
#> chr ""
```

If you are likely to want to include either single or double quotes themselves in your string, it's a good idea to use the other type of quotes for defining your string. I recommend using double quotes by default ("x"), and single quotes when you wish to include quoted text ('He said "Hello"').<sup>19</sup> Alternatively, you can specify that the quotes inside your string are not to be interpreted using a *slash* (\") to *escape* the character, so they won't end the string

```
"The sign said, \"Walk\"."
#> [1] "The sign said, \"Walk\"."
```

The mixture approach, such as

```
"The sign said, 'Walk'." 
#> [1] "The sign said, 'Walk'."
```

means we can use apostrophes in our text without issue

```
"That's John's father's name."
#> [1] "That's John's father's name."
```

With the alternative arrangement

```
'The sign said, "Walk".'
#> [1] "The sign said, \"Walk\"."
```

we see that the double quotes have been *escaped* (preceded by \) for us.



### Escape character

There are several **special characters** in R that have a particular meaning inside strings. These are entered by immediately preceding them with a backslash (\).

The ones you're most likely to encounter are

---

<sup>19</sup> Many editors will automatically insert the paired double quote for you when you enter ".

```
\'
single quote

\""
double quote

\n
newline

\\
backslash itself
```

When a string is merely printed, the escape character is also printed for clarity. The function `cat()` (short for 'concatenate', another word for join) interprets the special characters when it outputs something to the screen

```
cat('\\$pecial characters\\ndo \\$pecial things\\')
#> 'Special characters'
#> do 'special things'
```

Notice how the string is both defined with, and uses, single quotes, and the explicit line-break (using `\n`). Take caution when using a backslash in strings in case you unintentionally create a special character. If you require a backslash, it's best to escape it (`\\\`).

Behind the scenes, something similar is happening to when we store a number. A computer has no way to literally store the letter a, so again it is converted to 1s and 0s using an *encoding* scheme (basically a look-up table of values and their corresponding representations on the screen). There are even more ways to do this, and unfortunately there is no perfect consensus on how, so occasionally computers get confused about how to translate between the stored version and the version you see on the screen.



### Encodings

If you've ever received a garbled email that looks something like â € ™ then the encoding has somehow been incorrectly set. In this case, an apostrophe ('') is encoded using a scheme called UTF-8 as three *hexadecimal* (base 16, using 0-9 and A-F) values: 0xE2 0x80 0x99 (alternatively written as e28099), but those three values in the Windows-1252 encoding look-up table correspond to the symbols â, €, and ™, so if the computer gets confused about which scheme to use, the text appears different to what was expected.

We need not worry about how the computer does this for the most part, and R will happily work with the text you type in.

### 2.1.3 Categories (Factors)

Sometimes you want to work with categories of things; *categorical data* as opposed to *continuous data* which could be any value within some range. When working with categories, there are a limited number of choices from which your values might be taken, and it can be useful to let R know that this is the case.

### Categorical

When you have a limited set of things you are describing, you can consider them in their own distinct little boxes. One for each category. There exists some distinction between each of them that makes the categories meaningful: Country names for example are distinct, so these can form categories.

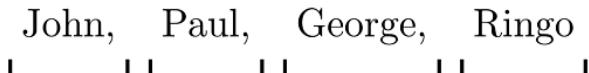
### Continuous

When you have a range of values without distinct differences between them, placing those values in meaningful boxes becomes tricky. Rather than creating smaller and smaller boxes, we can consider the entire range of values together. Consider the range of heights of a sample of adults; depending on how precise you are measuring, this could be any value between the shortest and tallest heights.

When the values are numbers (*numeric* or *integer*) but only certain values are to be considered, we call these *discrete values*. These might be the number of cylinders in an engine (4, 6, 8) or television screen dimension measurements (30, 36, 40, 52, 60). They don't need to be integers, they just need to have specific values (e.g. 3.2, 9.4, 7, 129.4). R will store their labels as character (text) either way.

Alternatively, you might have categories that are described by text, such as names ("John", "Paul", "George", "Ringo") or places ("Los Angeles", "New York", "Chicago", "Seattle").

When you have data like this and wish for R to know that there are only certain values available, you can convert the data to the type **factor** in which case R creates a look-up table of the names for your categories (either the numbers, text, or whatever type you have) and the category *index* (number, with some ordering).

John, Paul, George, Ringo  
  
 2            3            1            4

This allows R to only have to use the category *values* once, and can resort to the index whenever it needs to do something with it. This was once very useful in terms of memory saving, but that aspect has become less important as memory has become so easily available.

The ordering is typically alphabetical (by default) and we can see how R has chosen to do this by using **str()**. Enter the following into the Console, either line by line, or copy all of it and paste it all at once

```
str(object =  
  factor(x =  
    c("John", "Paul", "George", "Ringo")))  
)  
)  
#> Factor w/ 4 Levels "George", "John", ... : 2 3 1 4
```

- ➊ `str(object)` asks for the structure of object.
- ➋ `factor()` tells R to create a factor object from the values in the argument `x`.
- ➌ `c()` creates a *concatenated* (joined) group, which we'll see in Chapter 5.

In this case, the categories (the names) are ordered alphabetically, so "George" receives index 1, "John" receives index 2, "Paul" receives index 3, and "Ringo" receives index 4. This can be seen in the `str()` output which lists the factor levels then describes the data by these index values; 2 3 1 4.



#### On factor levels

**Be careful not to confuse the *index* values with the ordering:** in the above example the values "John", "Paul", "George", "Ringo" were assigned the alphabetical indices 2 3 1 4. This is not a re-ordering; the data still appears in the sequence we specified, but now the names have been replaced by a number. We can return to the original names by replacing the other way, so 1 becomes "George", and so on.

Factors can be very useful when you're dealing with categories because it *abstracts* away the category labels into their *index*. Sometimes, you want to preserve some information about those categories — the fact that they have some natural order. One example is when your categories are groups or ranges of values; "small", "medium", "large". In this case, you want to keep in mind that the "small" category represents things that are smaller than those in the "medium" category which themselves are smaller than those in the "large" category.

```
str(object = factor(x = c("medium", "small", "large")))
#> Factor w/ 3 Levels "Large","medium",...: 2 3 1
```

Note that the order (alphabetical) is the opposite of what you probably want.

In this case, you can tell R that this is an *ordered factor*, in which case it will use the ordering you specify in the `levels` argument

```
sizes <- factor(x      = c("medium", "small", "large"),
                 levels = c("small", "medium", "large"),
                 ordered = TRUE)
str(object = sizes)
#> Ord.factor w/ 3 Levels "small" <"medium" <...: 2 1 3
```

This becomes important when we want our outputs to display in a particular order:

```
table(sizes)
#> sizes
#>   small medium  large
#>   1       1       1
```

Factors can be a source of great frustration when they show up unexpectedly, so it's a good idea to know when and where R will try to help you out a bit too much and use them.



### Dial m for Male. Or M. Or male.

Even in many professional settings, data is often entered by hand. This can lead to conventions not being followed, and different names for things being used when a consistent value should be. One common example is entering of "male" and "female" identifiers. When R sees a text column that it needs to convert to a factor, it generates the full list of options and assigns the index of each occurrence. If most of your values in a 'sex' column are "male" or "female" but the occasional "m", "M", "male", or "Male" has slipped through, R will gladly — and silently — create this as a new factor level.

The best way to not get caught by this is to inspect the levels created whenever this might be an issue. The `levels()` function returns the unique levels of a factor variable.

```
levels(x = sizes)
#> [1] "small"   "medium"  "Large"
```

This is slightly different behaviour to Excel which is known to ignore upper/lower case when filtering values.

## 2.1.4 Dates and Times

When dealing with data that has been collected at (or describes) a particular date and time, we need to deal with these structures. Numbers aren't sufficient, and strings don't have any rules to them. R has several special types for dealing with dates and times.

When you have dates without times, the `Date` type is most useful, but if you also have times associated with those dates, the `POSIXct` type allows that functionality.

```
"2017-02-03"    → Date
"2017-02-03 23:16:59 CST" → POSIXct
```

We can examine examples of each of these structures by asking R to produce the current time or date as the system understands it to be using two built-in functions, `Sys.time()` and `Sys.Date()`. The former produces, as you may well guess, the current time, while the latter produces the date.

```
str(object = Sys.time())
#> POSIXct[1:1], format: "2018-01-23 21:38:47"
```

Here R tells us that the time is stored as type `POSIXct` and gives us an example of the format. We can also be less specific and request the date

```
str(object = Sys.Date())
#> Date[1:1], format: "2018-01-23"
```

Have you ever read the date as someone's written it and wondered what order they intended? Is `04-08-16` the 4th of August in 2016, the 8th of April 1916, or the 16th of August 2004? For the purposes of consistency and reproducibility, it is important to choose an order and stick to it. I highly recommend the ISO-8601 standard method of writing dates in the format of `YYYY-MM-DD` for year-month-day, regardless of what is more common where you happen to live. This means that dates can be sorted more easily, and will generally be easier to work with.

R requires that the input format be specified using percent signs and certain codes, which are described in another help file (`?strptime`). R uses the default format of "%Y-%m-%d %H:%M:%S" with the codes specified in Table 2.2. Note also that the capitalisation is important %m represents month, while %M represents minute.

**Table 2.2. Date and time input format codes.<sup>[a]</sup>**

Code	Meaning	Range
%Y	year (including century)	1 - 9999
%m	month	01 - 12
%d	day	01 - 31
%H	hour	00 - 23
%M	minute	00 - 59
%S	second	00 - 59

[a] For more, see `?strptime`.

There are many additional codes, for example to specify a month abbreviation such as "Feb" (%b) and these are listed in the help menu for the function `strptime`; obtained by evaluating `?strptime`.

Working with dates in this format is made easier with additional packages, but the most basic way to extract a part of a date or time object (say, the year) is to change the printed format to include the structure you wish to extract. If we want to extract the current year from a call to `Sys.time()` we can achieve this with the `format()` function which formats its input into a character string. It takes the same format codes as the date and time functions, so we can extract the year (as YYYY with %Y)

```
format(x = Sys.time(), format = "%Y")
#> [1] "2018"
```

An additional complication is the timezone that a `Date` or `POSIXct` value corresponds to. R is certainly able to handle this, but as a first precaution it's advisable to simply ensure that all of your data corresponds to a single timezone.

## 2.1.5 Logicals

The values `TRUE` and `FALSE` can be extremely useful, and as such these are reserved names in R, which means they can't be reassigned. R reports these as type `logi` (for 'logical', sometimes referred to as 'boolean')

```
str(object = TRUE)
#> Logi TRUE
```

These become highly useful when we wish to execute code *conditionally* based on some other calculation, which we'll cover in chapter 8.



### You can't handle the truth

For historical reasons, the values `T` and `F` also exist in R and have the values `TRUE` and `FALSE`, respectively. Unlike their full-named counterparts however, these aren't reserved names, so

they can be assigned (I highly recommend against using either of these as variable names for this reason).

If one felt evil enough, they could cause a great deal of confusion by setting these to their opposite value deep within some code then testing these values further down.

These are frequently used when you have binary data — which can only be in either of two states. Alive or dead, up or down, on or off, yes or no, or literally true or false. By storing this type of data as logical you ensure that if something has a value, it can only take one of these values.

Settings are also frequently stored with this type. We can turn on (or off) the printing of quote marks when we `print()` results

```
print(x = "abc")
#> [1] "abc"

print(x = "abc", quote = FALSE)
#> [1] abc
```

Sometimes however, while we wish for our data to be stored in this type, as either TRUE or FALSE, we simply don't know the answer. In that case, there is a third option which fits neither of these labels; we can explicitly specify that some data simply isn't there; it's missing.

## 2.1.6 Missing Values

R has several ways of denoting data that is missing. The most common of these is NA ('not available') which represents data that is skipped over or not present. While this value may be not present, R can still keep track of what *type* it was supposed to be. There are a few 'flavours' of NA, even though they're still missing data: NA\_real\_, NA\_integer\_, and NA\_character\_ are the ones you're most likely to encounter. By default, NA is a logical type.

```
typeof(x = NA)
#> [1] "Logical"
```



### Don't ignore what's missing

Missing values can be extremely important information, so take care about how you use them. It's not at all uncommon to see people producing spreadsheets with a mixture of empty cells, sentinel values (implausible values that signal something else, such as -99), zeroes, and text representations of missingness such as "N/A" or ". .".

Missingness might represent the fact that an observation/measurement wasn't taken/Performed, rather than it producing an actual value of 0. That distinction becomes very important when 0 is a meaningful value for the observation/measurement, such as a count of something. Many of R's statistical functions have special options for dealing with missing values that are coded as NA, typically the TRUE/FALSE option `na.rm` which instructs that a function should remove NA values before calculating.

NULL represents part of a structure which is not just missing but non-existent entirely;

the underlying structure of a `NULL` object is inherently different.<sup>20</sup> `NA` and `NULL` differ in that `NA` represents a value that could have existed but which doesn't, while `NULL` represents the lack of even the structure in which the value could have existed. `NaN` (not a number) refers to values which are undefined (such as  $0/0$ ) though these are rarely encountered.

In addition to these, the value `Inf` refers to the 'value' of infinity (such as produced by evaluating  $1/0$ ) which is larger than any other number. We can also refer to `-Inf` which is smaller than any number. This isn't really a missing number, but at the same time it isn't a number you can do much with. It can however be useful when comparing values (which we will see in chapter 3) as `Inf` is larger than any other value.

## 2.2 Storing Values (Assigning)

Up until now we have only been asking `R` for the *structure* of data, and we haven't particularly done anything with it. The function `str()` merely prints details about the structure of something to the screen, it doesn't assign (name and keep) any values for later use.

In order to do something with our data, we will need to tell `R` what to call it, so that we can refer to it in our code. In programming in general, we typically have *variables* (things that may vary) and *values* (our data). We've already seen that different data *values* can have different *types*, but we haven't told `R` to store any of them yet. Next, we'll create some *variables* in which to store our data *values*.

### 2.2.1 Naming Data (Variables)

If we have the values 4 and 8 and we want to do something with them, we can use the values literally (say, add them together as  $4 + 8$ ). You may be familiar with this if you frequently use Excel or some other spreadsheet software; data values are stored in cells (groups of which you can choose to name) and you tell the program which values you wish to combine in some calculation by selecting the cells with the mouse or keyboard. Alternatively, you can choose to refer to cells by their grid reference (e.g. `A1`). An example of this is shown in Figure 2.1.

**Figure 2.1. Spreadsheet formula to add two cell values.**

	A	B	C
1	4	8	=A1+B1
2	5	9	
3	6	10	

When the cell formula is evaluated, the result takes its place, as in Figure 2.2.

---

<sup>20</sup> Technically, there is only a *single* `NULL` object in any `R` session and any instances of variables given this value simply point to that.

**Figure 2.2. Spreadsheet formula result.**

	A	B	C
1	4	8	12
2	5	9	
3	6	10	

Similarly to the A1 and B1 references, we can store values in *variables* (things that may vary, also called *objects*) and *abstract* away the values. In R, assigning of values to variables takes the following form



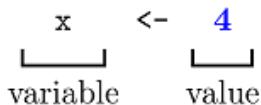
```
variable <- value
```

The *assignment operator* `<-` can be thought of as storing the value/thing on the right hand side into the name/thing on the left hand side.

Try typing the following into the R Console then press .

```
x <- 4
```

A diagram of the components involved in assignment is shown in Figure 2.3.

**Figure 2.3. Components of an assignment operation.**

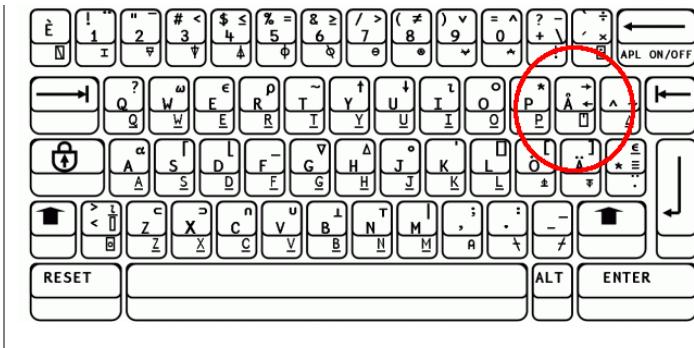
You could just as easily use the equals sign to achieve this; `x = 4` but I recommend you use `<-` for this for reasons that will become clear later. It may seem odd that multiple characters are required to type `<-` but you'll get used to it.



#### Why not just equals?

Back when the S language was developed, a popular computer keyboard was the APL keyboard which actually featured a `←` key, so this was even easier to input. An example is shown in Figure 2.4.

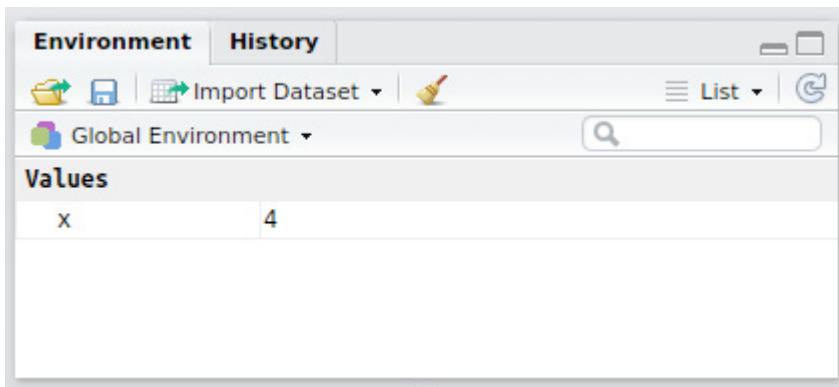
Figure 2.4. <https://commons.wikimedia.org/wiki/File:APL2-nappaimisto.png> Adapted from Wm313 Creative Commons Wikipedia.



If you're using RStudio (and I hope you are) then there is a shortcut to entering this, with default keys `Alt + -` (`Option + -` on Mac).

You'll notice that the **Environment** tab of the **Workspace** pane now lists `x` under **Values** and shows the number 4 next to it, as shown in Figure 2.5. Your **Environment** listing may show more variables, depending on what else you've assigned in your session so far.

**Figure 2.5. Environment pane showing the value 4 assigned to the variable x.**



What happened behind the scenes was that when we pressed `Enter`, R took the entire expression that we entered (`x <- 4`) and evaluated it. Since we told R to *assign* the value 4 to the variable `x`, R converted the value 4 to binary and placed that in the computer's memory. R then gives us a reference to that place in the computer's memory and labels it `x`. A diagram of this process is shown in Figure 2.6. Nothing else appeared in the **Console** because the action of assigning a value doesn't *return* anything (we'll cover this more in our section on functions).

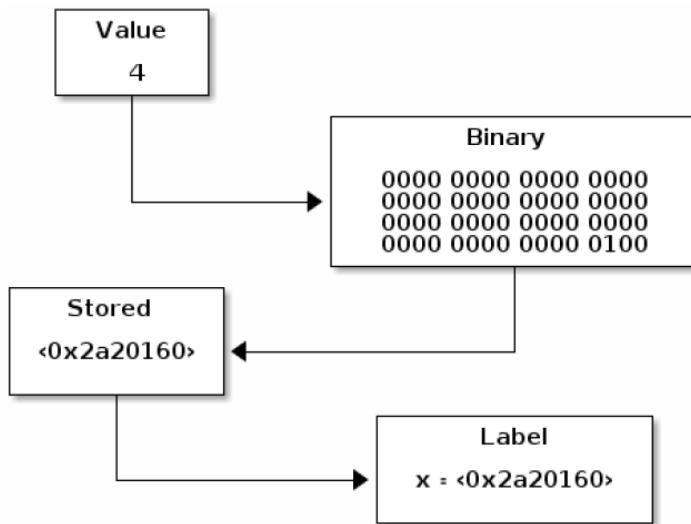


If you really want to force the printing of a value during assignment you can do so by surrounding a non-returning (or any) expression with parentheses, i.e. normally an assignment prints nothing

```
no_return <- 7
but we can force it to if we start the expression with a ( and end with a )
(return_me <- 7)
#> [1] 7
```

This is particularly handy when you wish to show what you've just assigned, either for testing or demonstration. This should be restricted to single lines, as this won't work on multi-line expressions.<sup>21</sup>

**Figure 2.6. Assigning a value to a variable. The value entered is converted to binary, then stored in memory, the reference to which is labelled by the variable.**



This is overly simplified, of course. Technically speaking, in R, names have objects rather than the other way around. This means that R can be quite memory efficient since it doesn't create a copy of anything it doesn't need to.



#### On "hidden" variables

Variables which begin with a period (e.g. `.length`) are considered *hidden* and do not appear in the Environment tab of the Workspace. They otherwise behave exactly as any other variable; they can be printed and manipulated. An example of one of these is the `.Last.value` variable, which exists from the moment you load up R (with the value TRUE) — this contains the output value of the last statement executed (handy if you forgot to assign it to something). There are very few reasons you'll want to use this feature (dot-prefixed hidden variables) on purpose at the moment, so for now, avoid creating variable names with this pattern. You can request to be visible in the Environment tab via Tools → Global Options... → General → Show .Last.value in environment listing.

<sup>21</sup> You may be able to also surround an expression in curly braces *then* parentheses to allow this, but at that point you should really wonder why you need that.

We can retrieve the value assigned to the variable `x` by asking R to print the value of `x`

```
print(x = x)
#> [1] 4
```

for which we have a useful shortcut — if your entire expression is just a variable, R will assume you mean to `print()` it, so

```
x
#> [1] 4
```

works just the same.

Now, about that `[1]`: it's important to know that in R, there's no such thing as a single value; every value is actually a *vector* of values (we'll cover these properly in the next chapter, but think of these as collections of values of the same type).<sup>22</sup> Whenever R prints a value it allows for the case where the value contains more than one number. To make this easier on the eye, it labels the first value appearing on the line by its position in the collection. For collections (vectors) with just a single value, this might appear strange, but this makes more sense once our variables contain more values. For example, if we print the column names of the mtcars dataset

```
colnames(x = mtcars)
#> [1] "mpg"   "cyl"   "disp"  "hp"    "drat"  "wt"    "qsec"  "vs"    "am"    "gear"
#> [11] "carb"
```

We can assign another value to another variable

```
y <- 8
```

We can create as many variables as we need, as long as they fit within the available memory. All R objects are stored within the computer's RAM, so this limits the size of these objects.<sup>23</sup>

There are few restrictions for what we can name our *variables*, but R will complain if you try to break them. Variables should start with a letter, not a number. Trying to create the variable `2b` will generate an error

```
2b <- 7
```



**Error: unexpected symbol in "2b"**

Variable names can start with a dot (.) as long as it's not immediately followed by a number, although you may wish to avoid doing so (these are 'hidden' variables, but otherwise work as normal). The rest of the variable name can consist of letters (upper and lower case) and numbers, but not punctuation (except . or \_) or other symbols (except the dot, though again, preferably not).

<sup>22</sup> In technical terms, R has no *scalar* values.

<sup>23</sup> You would need to create a *lot* of large variables to exhaust even a modest amount of RAM on a modern computer.



### Quote-defined variables

If you absolutely need to bypass these restrictions you can surround your variable name in quotes or backticks when assigning, and refer to it with backticks when using it. There are very few situations in which this will be a good idea, but it's good to know it's possible. For example, starting with a number

```
"2b" <- 7
7 + `2b`
#> [1] 14
```

or including spaces

```
`to be or not to be` <- 2
7 + `to be or not to be`
#> [1] 9
```

Trying to access these with quotes won't work (you just get the character string)

```
"2b"
#> [1] "2b"
```

they must be referenced with backticks

```
`2b`
#> [1] 7
```

This will become important later when we read in external data.

There are also certain *reserved words* that you can't use as variable names. Some are *reserved* for built-in functions or keywords, most of which we'll cover later

`if, else, repeat, while, function, for, in, next, and break.`

Others are *reserved* for particular values

`TRUE, FALSE, NULL, Inf, NaN, NA, NA_integer_, NA_real_, NA_complex_, and NA_character_.`

and we've covered what each of these means, so hopefully you can see why you can't create a variable with one of those names.



### On overwriting names

What you **can** do however, which you may wish to take care with, is *overwrite* the built-in names of variables and functions. By default, the value `pi` is available ( $\pi = 3.141593$ ).

If you were translating an equation into code, and wanted to enter a variable referring to the *i*th value of `p`, `pi` you might accidentally call it `pi` and in doing so change the default value, causing all sorts of trouble when you next go to use it or call a function you've written which expects it to still be the default.

The default value can still be accessed by specifying the package in which it is defined, separated by two colons (`::`). In the case of `pi`, this is the base package.

```
pi <- 3L ①
base::pi ②
#> [1] 3.141593
```

- ➊ Re-defining pi to be equal to exactly 3
- ➋ The default, correct value is still available.

This is also an issue for functions, with the same solution; specify the package in which it is defined to use that definition. We'll return to this in [Scope](#).

We'll cover how to do things to our variables in more detail in the next section, but for now let's see what happens if we add our variables `x` and `y` in the same way as we did for our regular numbers

```
x + y
#> [1] 12
```

which is what we got when we added these numbers explicitly. Note that since our expression produces just a number (no assignment), the value is printed. We'll cover how to add and subtract values in more depth in [Basic Mathematics](#).

R has no problems with overwriting these values, and it doesn't mind what data you overwrite these with.<sup>24</sup>

```
y <- "banana"
y
#> [1] "banana"
```

R is *case-sensitive*, which means that it treats `a` and `A` as distinct names. You can have a variable named `myVariable` and another named `MYvariable` and another named `myVARIABLE` and R will hold the value assigned to each independently.

### **On variable names**

**“ There are only two hard things in Computer Science: cache invalidation and naming things.”**

-- Phil Karlton *Principal Curmudgeon Netscape Communications Corporation*

I said earlier that R won't keep track of your units so it's a good idea to name your variables in a way that makes logical sense, is meaningful, and will help you remember what it represents. Variables `x` and `y` are fine for playing around with values, but aren't particularly meaningful if your data represents speeds, where you may want to use something like `speed_kmph` for speeds in kilometers per hour. Underscores (`_`) are allowed in variable names, but whether or not you use them is up to you. Some programmers prefer to name variables in this way (sometimes referred to as 'snake\_case'), others prefer 'CamelCase'. The use of periods (dots, `.`) to separate words is discouraged for somewhat more advanced reasons.<sup>25</sup>

---

<sup>24</sup> This is where the distinction of *weakly typed* becomes important - in a *strongly typed* language you would not be able to arbitrarily change the type of a variable.

<sup>25</sup> This syntax is already used within R to denote functions acting on a specific class, such as `print.Date()`.



### Naming things

Be careful when naming your variables. Make them meaningful and concise. In six months from now, will you remember what `data_17` corresponds to? Tomorrow, will you remember that `newdata` was updated twice?

## 2.2.2 Unchanging Data

If you're familiar with working with data in a spreadsheet program (such as Excel), you may expect your variables to behave in a way that they won't. Automatic recalculation is a very useful feature of spreadsheet programs, but it's not how R behaves.

If we assign our two variables, then add them, we can save that result to another variable

```
a1 <- 4
a2 <- 8
sum_of_a1_and_a2 <- a1 + a2
```

This holds the value we expect

```
print(x = sum_of_a1_and_a2)
#> [1] 12
```

and is the same result we would get from using, say, `SUM(A1:A2)` in a spreadsheet, as shown in Figure 2.7.

**Figure 2.7. Adding two values in a spreadsheet.**

	A
1	4
2	8
3	=A1+A2

	A
1	4
2	8
3	12

Now, if we *change* one of these values in R

```
a2 <- 7
```

this has **no impact** on the value of the variable we created to hold the sum earlier, or any other variables in the workspace

```
print(x = sum_of_a1_and_a2)
#> [1] 12
```

whereas a spreadsheet would most likely recalculate the sum. Note also that the value of `sum_of_a1_and_a2` shown in the Environment pane hasn't changed when we reassigned `a1` or `a2`.

Once the sum was calculated, and that value stored in a variable, the connection to the original values was lost. This makes things *reliable* because you know for sure what value a variable will have at any point in your calculation by following the steps that

lead to it, whereas a spreadsheet depends much more on its current overall *state*.

### 2.2.3 The Assignment Operators (<- vs =)

If you've read some R code already, you've possibly seen that both <- and = are used to assign values to variables, and this tends to cause some confusion. Technically, R will accept either when assigning variables, so in that respect it comes down to a matter of style (I still highly recommend assigning with <-). The big difference comes when using functions that take *arguments* — there you should only use = to specify what the *value* of the *argument*. For example, when we inspected the mtcars data, we could specify a string with which to indent the output

```
str(object = mtcars, indent.str = "INDENT>> ")
#> 'data.frame':      32 obs. of  11 variables:
#> INDENT>> $ mpg : num  21 21 22.8 21.4 18.7 ...
#> INDENT>> $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
#> INDENT>> $ disp: num  160 160 108 258 360 ...
#> INDENT>> $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
#> INDENT>> $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
#> INDENT>> $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
#> INDENT>> $ qsec: num  16.5 17 18.6 19.4 17 ...
#> INDENT>> $ vs   : num  0 0 1 1 0 1 0 1 1 1 ...
#> INDENT>> $ am   : num  1 1 1 0 0 0 0 0 0 0 ...
#> INDENT>> $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
#> INDENT>> $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

If we had used <- instead of = for either argument, then R would treat that as creating a new variable object or indent.str with value mtcars or ">>" respectively, which isn't what we want.

 Have a play with assigning some variables yourself and see if you always get the results you expect:

#### Listing 2.1. Try it yourself: assigning to variables.

```
score <- 4.8
score
#> [1] 4.8

str(object = score)
#> num 4.8

fruit <- "banana"
fruit
#> [1] "banana"

str(object = fruit)
#> chr "banana"
```

Note that we didn't need to tell R that one of these was a number and one was a string, it figured that out itself. It's good practice (and easier to read) to make your <- line up vertically when defining several variables:

```
first_name <- "John"
last_name  <- "Smith"
```

```
top_points <- 23
```

but only if this can be achieved without adding too many spaces (exactly how many is too many is up to you).



#### Watch this space

An extra space can make a big difference to the syntax. Compare:

```
a <- "x"
```

with

```
a < - 3 ①
#> [1] FALSE
```

① < is the 'less than' operator.

In the first case we assigned the value "x" to the variable a (which returns nothing). In the second case, with a wayward space, we compared a (which now holds the value "x") to the value -3 which returns FALSE (I'll explain why that works at all, when we look at comparing values in ["Automatic Conversion \(Coercion\)"](#)).

Now that we know how to provide some data to R, what if we want to explicitly tell R that our data should be of a specific type, or we want to convert our data to a different type?

## 2.3 Specifying the Data Type

By default, R will assume the most general type for your data as you assign it to a variable. We've seen that assigning a value that only *looks* like an integer will still be stored as a numeric type, just in case

```
myInt <- 3
str(object = myInt) ①
#> num 3
```

① This is actually a numeric type.

unless we specifically tell R otherwise

```
myInt <- 3L
str(object = myInt) ①
#> int 3
```

① NOW it's an integer.

Sometimes though, your data isn't of the *type* that you want it to be. So, how do we convert it? R has a group of functions that start with `as.` which attempt to convert any input data (*coerce*) into another type. The important ones at this point are:

- `as.integer()`
- `as.numeric()`
- `as.character()`

- `as.logical()`
- `as.POSIXct()` and `as.POSIXlt()`
- `as.Date()`

These take whatever data you have and convert it to a specific *type*. So, if we have a number, but we want R to treat it like text (character/string) we can use `as.character()`

```
numAsString <- as.character(x = 1234)
str(object = numAsString)
#> chr "1234"
```

We can of course force this when entering the data by placing quotes around the number, making it a string

```
numAsString <- "1234"
```

but sometimes we wish to convert a value already stored in a variable

```
num <- 1234
numAsString <- as.character(x = num)
str(object = numAsString)
#> chr "1234"
```

Again we see that it becomes difficult (impossible) to distinguish between numeric and integer types once converted to character strings

```
as.character(x = 7)
#> [1] "7"
as.character(x = 7L)
#> [1] "7"
```

More common perhaps would be to go the other way; to treat a character string as a number. Several functions will do this for you by default, but often you'll need to do the conversion yourself. If you have some numeric data that has been imported as character strings, you can convert these back to numbers with

```
stringAsNum <- as.numeric(x = "3.14159")
str(object = stringAsNum) ①
#> num 3.14
```

① Now numeric.

Don't worry that the `str()` function reports fewer decimal places, the entire value is still there with as many digits as we provided

```
print(x = stringAsNum)
#> [1] 3.14159
```

Where this becomes slightly more difficult is when dealing with dates. Sometimes (or perhaps more often than not) your dates won't be in the default format of "%Y-%m-%d", in which case you'll need to use the optional `format` argument to specify this according to the codes in [Date and time input format codes](#).

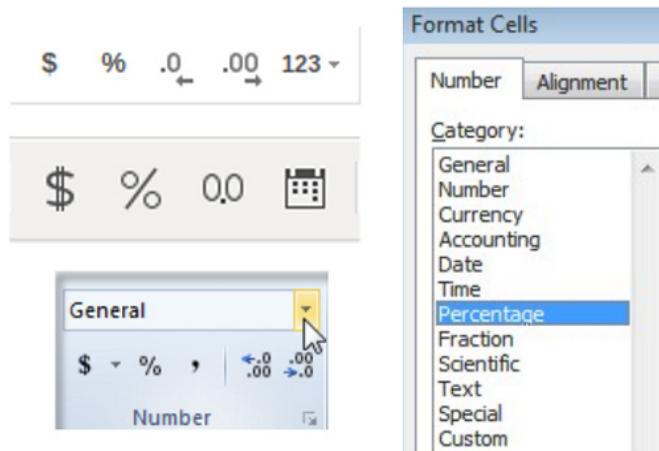
When telling R that a number is an integer rather than a numeric type, there is an alternative to the L-suffix syntax, and that is to explicitly tell R that our number is an integer

```
myInt <- as.integer(x = 7)
str(object = myInt) ①
#> int 7
```

① Also an integer.

This has some similarities to the conversion mechanism of spreadsheets, which provide a menus of data types for conversion, some examples of which are shown in Figure 2.8.

**Figure 2.8. Various ways to format numbers in different spreadsheet programs.**



There are limits to what R will convert. What if we provide something that clearly *isn't* an integer? You can test this with a few values yourself and see if you get the result you expect

```
as.integer(x = 3.14)
#> [1] 3
```

Makes sense perhaps. That's certainly the nearest integer (whole number) to that value. Let's try another

```
as.integer(x = 2.99)
#> [1] 2
```

Is that what you expected? Depends on how you guessed it might work. `as.integer()` *coerces* to the closest integer by rounding towards zero.



#### Rounding towards zero

#### What about negative numbers?

```
as.integer(x = -22.6)
#> [1] -22
```

Again the rounding is towards zero. This may or may not be what you want to do, so make sure you keep this rule in mind when using this function.

R will do its best to coerce from any type to any other type you ask, but only certain combinations will be possible. Integer to numeric is straightforward. Numeric to integer, as we just saw, works provided you know that it will round towards zero. What about some other types? What about requesting the integer version of a character string involving a negative sign and some unnecessary whitespace?

```
as.integer(x = " -47.8 ") ①
#> [1] -47
```

- ① Note the included whitespace.

One might not be too surprised that a clever enough system could figure this out. As soon as there is something unclear about the string however

```
as.integer(x = "--92.4")
# Warning message:
# NAs introduced by coercion
#> [1] NA
```

the conversion fails. This would usually be understood as applying the *unary* (acting on one variable) function of 'take the negative value' twice, if it weren't a string

```
as.integer(x = --92.4)
#> [1] 92
```

NA (for *Not Available*) is the value produced whenever a conversion isn't possible or a value is missing, and R will usually give you a warning to let you know it's done that.

`as.character()` can convert just about anything, since all it needs to do is throw some quotes around the input

```
as.character(x = 2342.983)
#> [1] "2342.983"
```



### Factor labels

Be careful if you are trying to extract factor levels using these functions. Say we have our factor levels

```
dozens <- factor(x = c(36L, 24L, 12L, 48L))
dozens
#> [1] 36 24 12 48
#> Levels: 12 24 36 48
```

and we wish to go back to just integer types. While it's reasonable to expect that you could just use `as.integer()`, something goes horribly wrong

```
as.integer(x = dozens)
#> [1] 3 2 1 4
```

R has given us the factor *indices*, not the actual levels. Instead, we need to *first* convert our factor to character to obtain the levels, then we can convert to integer

```
as.integer(x = as.character(x = dozens))
#> [1] 36 24 12 48
```

Note also that once the values are in factor form, they show no distinction between originally being numeric or integer,<sup>26</sup> so there's no way to explicitly 'convert back' to their original type.

R considers certain values to be convertible to TRUE or FALSE via `as.logical()`. Non-zero numbers convert to TRUE, while 0 converts to FALSE. This perhaps makes sense in the context of a signal being either on or off.

```
as.logical(x = 0)
#> [1] FALSE

as.logical(x = 1)
#> [1] TRUE
```

Most characters/strings convert to NA, the missing value

```
as.logical(x = "value")
#> [1] NA
```

with some notable exceptions; the strings T, TRUE, True, and true, as well as F, FALSE, False, and false each convert to their respective logical values, which can make for some odd results

```
as.logical(x = "R")
#> [1] NA

as.logical(x = "S")
#> [1] NA

as.logical(x = "T")
#> [1] TRUE
```

We can see the difference between Date and `POSIXct` by explicitly converting the current system time (which naturally returns an object of type `POSIXct` anyway) into each of these types

```
as.Date(x = Sys.time())
#> [1] "2018-01-23"

as.POSIXct(x = Sys.time()) ①
#> [1] "2018-01-23 21:38:47 ACDT"
```

- ① The timezone is included in the output. The zone shown here (Australian Central Daylight Time, ACST) may very well be different to your own output in your timezone.

---

<sup>26</sup> The levels are stored as characters, and as we've seen, numeric and integer types look the same when converted to characters.



In the last example, `as.POSIXct()` (or just `Sys.time()`) has shown us the date, time (including seconds), and timezone (Australian Central Daylight Time, ACDT, in my case) for the current computer date-time. This will be different whenever/wherever you repeat it.

If your computer doesn't have the date, time, or timezone set correctly, this may not accurately reflect these quantities.

## 2.4 Telling R to Ignore Something

One of the most important components of any piece of programming, be it a full program or just a small snippet of code, is *documentation*. Most programming languages allow for *inline documentation* in the form of *comments*, which are ignored by the system (recall the way that R 'reads' your commands; [Read, Evaluate, Print, Loop](#)) but which are very useful to anyone reading the code.

R will ignore anything appearing after a # (including the #) on a line, so you can use this space to document **why** you've made the choice to write your code the way you have. Don't waste your time merely explaining **what** the code does — that should either be obvious (from reading the code itself) or should be justified with a **why** if it's not.

Exactly where you put your comments is a matter of style, but you should aim to keep them as close to the relevant piece of code as possible. You don't need to have a comment for every line, but dividing your code into meaningful blocks can help the person reading it (even if that's you).

You may wish to start a new line with a # which will mean that the entire line will be ignored by R and will contain comments. If your comments are too long for the line and *wrap-around* to a new line, make sure that this too starts with a #. RStudio will often help you by filling this in for you.

Comments can also come part-way through a line, in which case R will ignore everything from the # onwards. This can be handy when you want to make a quick note on a single command, but having too-long a line can become difficult to read. When these lines wrap-around, the continued comment needs to start with a # also. R has no way to mark a large block of comments across several lines, so each line will need to include its own #. RStudio can help with this though; if you highlight several lines then press `Ctrl + Shift + C`, the selected lines will all be commented by inserting a # at the start of those lines (even if it's just one).

Since comment code is ignored, this feature can also be used to 'turn off' parts of your analysis code. It can be tempting to rely on this too much, resulting in a large amount of disabled code spread throughout your scripts. It is much better to delete any code you are no longer consider 'active'. Making good use of [Version Control](#) allows you to *roll-back* to older version of your scripts if you want to bring back in features you deleted.

**Listing 2.3. Try it yourself: Documenting using comments**

```
# Here we are using a=2 because it provides the
# best agreement with the data. ①
a <- 2

# cosmological constant:
# ratio of dark energy pressure to energy per unit volume
w <- -1

y <- w + a # TODO: cross-check this with formula in Einstein (1917) ②

z <- log(x = y) # values of y are assumed to be positive and
# non-zero at this point, so this should be
# safe. ③

#x <- z - 1 ④
```

- ① Note that each comment starts with a #.
- ② Comments following code should rarely flow over more than one line, but can be made to look neat with careful use of whitespace.
- ③ Comments can serve as useful notes to yourself. Be sure to remove them if they aren't to be shared.
- ④ Inactive (*commented out*) code should not appear in your final analysis script, but can be a useful way to temporarily remove some code. This line will not be processed (it will be ignored by R).

## 2.5 Try It Yourself

 Choose a *type* from those we've discussed then enter a value into the Console. Press **Enter** and observe the result that appears. Repeat this for more *types* and take note of any differences you can find.

Wrap some values in `str()` and see how R describes these. Wrap more in `typeof()` and observe the different types. Can you create an integer and assign it to a variable called `my_int`? Can you change it to numeric type and assign it? Should you use a new name? Check the type you've created with `typeof()`.

How many R-acceptable ways can you write 6 billion?

See if you can create a *string* that R **won't** accept. Can you figure out why it isn't allowed? Can you fix it?

Can you assign the current date and time to a variable called `now`? Can you make it only show the month abbreviation and day?

Have a play with as many of the data types we've seen as you can and see if you can make anything surprising happen.

## 2.6 Summary

In this chapter we've seen how to tell R all about our different types of data, and how to assign values to variables. We've also seen how to change the type of those values

should we need to.

You've learned that:

- The computer stores all of its data in binary
- R has different types of data which it encodes differently
- R is weakly-typed, so you can change variable types if you need to
- Integers are treated as a distinct type of number, and can be entered with an L suffix
- Carefully naming your variables makes them easier to work with and recall
- Both types of quotes (single and double) work for working with strings
- You can escape characters by prepending them with a slash (\)
- Factors are stored as integers, but have character levels
- R is case-sensitive so "AbC" and "aBc" are distinct
- Dates and times can be stored as Date or POSIXct classes
- Missing values are denoted as NA for Not Available
- Assignment is performed using the <- operator
- Variable names can't start with a number
- Comments begin with a # and can appear at any point on a line

New terms you've learned:

#### ***real number***

a number with a whole and a fractional part (e.g. 3.14). R sometimes refers to these as 'double' or 'numeric'.

#### ***integer***

a number with only a whole part, specified by an L suffix (e.g. 5L).

#### ***character***

a text element (e.g. "a") containing zero or more symbols. R uses this name as the type of text objects.

#### ***string***

a series of zero or more characters, possibly including spaces (e.g. "I'm learning R!").

#### ***logical***

a 'truthy' value; TRUE or FALSE (or NA).

#### ***special character***

characters which are interpreted in a special way inside strings.

***escape character***

a slash (\) used before a special character, to specify that it should be interpreted differently.

***factor***

a labelled category level, used when the thing it labels can only take on a certain set of values.

***coerce***

the act of changing the type of a variable, potentially automatically.

Things to remember:

- It's best to explicitly write numbers as integers if that's what you're intending (the L suffix).
- Use double quotes ("x") for strings by default, unless you really need a quote inside a quote (in which case, use single quotes ' "always" ').
- Backslashes (\) are used to give special meaning to certain characters in strings.
- Factors are useful but sometimes show up unexpectedly. Use `as.character()` to extract the labels, not the indices.
- Prevent R from executing code with the comment character (#). Use this to write comments so you remember why you wrote the code the way you did.

Now it's time to learn how to combine some of these data values together to create new values.

# I Want To Make New Data Values



## This chapter covers:

- How to perform operations between two or more data values
- Comparing values of the same or different type
- How R changes the data type as it needs to

You have your data values, but there's a good chance you'll want to do something with them, like add or multiply. It's time to go back to basics and see how R deals with combining data. Thankfully, R is a readable language for things like this, and with any luck you'll be able to code up what you're trying to do with the operators you expect. Follow along this section in the `Console` and try some values yourself to see if you get the results you expect.

## 3.1 Basic Mathematics

The simplest thing you might want to do to two values is add them. No surprises here, the `+` symbol (*operator*) between two values will do just that, the same as you would on a calculator

```
2 + 2
#> [1] 4
```

The same goes for subtraction; no surprises here

```
4 - 3
#> [1] 1
```

Multiplication in programming tends to use the *asterisk* (\*) rather than a multiply sign (technically, `*`, but commonly seen as an `x`)

```
3 * 6
#> [1] 18
```

Division uses a slash (/) like you might use in writing a fraction

```
12 / 4
#> [1] 3
```

One that might not be so obvious is the *exponentiation* operator (raise a number to a power). This one is used for example when you need to square a value. There are two options here, though in reality they're just different ways of writing the same thing

```
2 ^ 10
#> [1] 1024
```

or, much less commonly, and potentially confused with multiplication on too-fast a skim-read,

```
2 ** 10
#> [1] 1024
```

Notice that no parentheses (()) were needed to group together the digits 1 and 0 into a 10 — R does some guesswork behind the scenes to interpret what you mean, and treats this as the value 10 rather than some invalid expression. The spaces between values and operators are also cleverly interpreted; we could just as easily remove them if we wanted and write 3+2 or 3^2 but for the sake of clarity, it's often best to space things out a little. The less-common variant \*\* does however require that there are *no* spaces between the two asterisks.

Dates are a special *type* in R (recall [Dates and Times](#)) and we can now see why; while you can certainly subtract dates to calculate a period of time, which R interprets sensibly for suitably encoded values

```
as.POSIXct(x = "2016-12-31") - as.POSIXct(x = "2016-01-01")
#> Time difference of 365 days
```

adding dates simply doesn't make sense

```
as.POSIXct(x = "2016-12-31") + as.POSIXct(x = "2016-01-01")
```



**Error: binary '+' is not defined for "POSIXt" objects**

Somewhat related, R won't let you perform operations between incompatible types for a given operator. While you might understand the intention of the following perfectly well

```
2 + "2"
```



**Error: non-numeric argument to binary operator**

we see that R won't take the leap of faith in assuming that you mean to add these as numbers. A spreadsheet may very well have the same apprehensions about adding a number to a string, an example of which is shown in Figure 3.1.

**Figure 3.1. Attempting to add a number to a string in a spreadsheet resulting in an error.**

The diagram shows a spreadsheet with two states. On the left, cell A1 has value 1, cell A2 has value 2, and cell A3 has formula '=A1+A2'. An arrow points to the right, where cell A3 now displays the error '#VALUE!'. The other cells remain the same.

	A
1	1
2	"2"
3	=A1+A2

	A
1	1
2	"2"
3	#VALUE!

Spreadsheets may however be more flexible when attempting this within a formula, as in Figure 3.2.

**Figure 3.2. Attempting to add a number to a string within a formula in a spreadsheet failing to result in an error.**

The diagram shows a spreadsheet with two states. On the left, cell A1 has formula '=2 + "2"'. An arrow points to the right, where cell A1 now displays the result '4'. The other cells remain the same.

	A
1	=2 + "2"

R is less fussy over the difference between numeric and integer types, and will allow you to work with these interchangeably if you ask, returning the result in the most general type possible (in this case, numeric which is more general than integer)

```
numPlusInt <- 7 + 3L
str(object = numPlusInt)
#> num 10
```

It should come as no surprise then that certain combinations are simply out of the question

```
"7" - "4"
```



Error: non-numeric argument to binary operator

```
"7" + "4"
```



Error: non-numeric argument to binary operator

while others may surprisingly work just fine

```
as.POSIXct(x = "2016-12-31") + 1 ①
#> [1] "2016-12-31 00:00:01 ACST"
```

- ① Remember that the timezone shown here (Australian Central Daylight Time, ACST) is likely to be different to your own output in your timezone.

Here R has treated the 1 as some number of *seconds* to add to the date-time object on the left side of the + operator. If we wish to add a whole day, then we need to add 24 hours, which each have 60 minutes, which each have 60 seconds, so

```
as.POSIXct(x = "2016-12-31") + 60*60*24 ①
#> [1] "2017-01-01 ACDT"
```

- ① While we certainly mean to be multiplying integers; which we would specify with an L suffix; we can be lazy and multiply numeric values and get the same result.



### Math with NA

Involving the missing data value NA typically leads to more missing data. Trying to add missing data to known data results in the total value going missing

```
7 + NA
#> [1] NA
NA + 0
#> [1] NA
```

This can lead to some unwanted effects if you have a lot of data values and just one missing value, perhaps because it was converted from another type incorrectly. The `sum()` function, which calculates the sum of its inputs, has an option specifically for this scenario (as do several functions). If we try to take the sum of some values including an NA value

```
sum(3, 7, 0, 9, NA)
#> [1] NA
```

the result is missing, but if we tell this function to first remove any NA values with the option `na.rm = TRUE`

```
sum(3, 7, 0, 9, NA, na.rm = TRUE)
#> [1] 19
```

the NA value was removed before the sum was calculated.

It's best to be suspicious any time you may have NA values in your data.

## 3.2 Operator Precedence

When we have multiple operations acting together, R has rules which determine the order in which they will be evaluated; their *precedence*

### Precedence

The order or ranking of a group. In R this refers to which operations will be performed before which others

For example, perhaps you've seen the 'challenges' floating around social media; where "99% of people won't get this right" is posted alongside a short and semi-ambiguous mathematical statement, such as



```
2 + 3 * 0 - 1 = ?
```

These should be no hassle for anyone who remembers their order of operations from

school; the acronym PEMDAS stands for **P**arentheses, **E**xponents, **M**ultiplication and **D**ivision, and **A**ddition and **S**ubtraction, which is the order in which these operations should be carried out. In the above case, given the lack of parentheses (()) or exponents (raised to a power) the next step is to perform the multiplication of 3 and 0 which is 0. This leaves the addition and subtraction steps which don't depend on their order, leaving



```
2 + 0 - 1 = 1
```

Nonetheless, the comments on said 'challenge' will contain any number of answers and people fighting tooth and nail to defend their incorrect justifications.

R has a similar hierarchy for order of operations, but it includes some other operators we've not discussed yet. Nonetheless, it's good practice to include parentheses where there is doubt about what the intended order of operations is, or you wish to force it to a particular grouping. In the case of the above, writing this as



```
2 + (3 * 0) - 1
```

would help make it clearer that the 3 multiplies the 0 and the result of that takes part in the subsequent addition and subtraction. This becomes essential when you wish to exponentiate to the result of some combination of multiple values



```
2 ^ (5 * 2)
```

where the **P**arentheses dictate that this group needs to be evaluated first, before the exponentiation. This can significantly alter the way that an expression is evaluated. Consider for a moment the following odd looking expression

```
x <- y <- 3 + 1
```

This is perfectly valid, and R will process it as it understands it (assign the result of 3 + 1 to y and assign that result to x), producing

```
x  
#> [1] 4
```

```
y  
#> [1] 4
```

Including some more parentheses though

```
x <- (y <- 3) + 1
```

we can change what the expression means (now assign 3 to y, then add 1, then assign that result to x) producing

```
x  
#> [1] 4
```

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

The computer (and R) has no problems understanding what you've written (a certain way). Do your best to make sure that what you've written is what you mean.

### 3.3 String Concatenation (Joining)

We saw earlier that adding two strings ("7" + "4") produced an error, because the + operator works for numbers (numeric or integer). We could envisage trying to 'add' two words together though, perhaps "butter" and "fly". We don't really mean 'add' though, we mean 'join', or more strictly, 'concatenate'.

For these circumstances, there is a specific function to perform this operation, the `paste()` function, but it has a default of joining strings with a space between them (which is often what you want)

```
paste("butter", "fly")
#> [1] "butter fly"
```

This is the result of the default argument `sep = " "` which specifies the separator to place between the inputs, with a default of using a space. We can change this to `sep = ""` to remove it entirely, or use the convenience function

```
paste0("butter", "fly")
#> [1] "butterfly"
```

which performs the same operation but with 0 spaces between inputs.

The `paste()` (and `paste0()`) function converts its input to character before pasting together, so we can use other *types* here too if we wish

```
paste0("value", 31)
#> [1] "value31"
```

or even variables we've defined, or other `paste()` calls

```
address_number <- 221
address_suffix <- "B"
address_street <- "Baker Street"

paste(
  paste0(
    address_number, ①
    address_suffix ①
  ),
  address_street ②
)
#> [1] "221B Baker Street"
```

- ① Inputs to the innermost `paste0()`
- ② Input to the outermost `paste()`



### Joining NA values

This is one scenario in which a missing value can become non-missing, perhaps unexpectedly. By default, the inputs to `paste()` are converted to type character with the `as.character()` function, but that function preserves missing values, so these two are different results

```
as.character(x = NA)    ①
#> [1] NA
as.character(x = "NA") ②
#> [1] "NA"
```

- ① a missing value
- ② the string "NA"

One might expect then that `paste()` will produce a missing value when one of its inputs is NA, but it handles this smoothly

```
paste("a missing value is denoted", NA)
#> [1] "a missing value is denoted NA"
```

This is noteworthy enough to receive special mention in the `help()` page for `paste()`:

#### From ?paste

Note that `paste()` coerces `NA_character_`, the character missing value, to "NA" which may seem undesirable, e.g., when pasting two character vectors, or very desirable, e.g. in `paste("the value of p is ", p)`.

Enter some more values into the Console with combinations of these operators and make sure you're comfortable with these. Next, we'll see how to make comparisons between values.

## 3.4 Comparisons

The essence of scientific results boils down to comparing values. Are there fewer `y` than a decade ago? Has `z` grown this week? Is `j` faster than `k`? Are any of `m` significant effects (fitting below some criteria)? If we didn't require comparisons between data, coding analyses would be fairly straightforward — always do this, then that, then the other thing. Since we do though, we should learn how to tell R to compare values.

The result of a comparison will always be a *logical* value (recall: [Logicals](#)); either `TRUE` or `FALSE` (or missing; `NA`). The simplest comparisons we can make are "is `x` greater than `y`?" (`x > y`) and "is `x` less than `y`?" (`x < y`) using these 'greater than' and 'less than' operators, `<` and `>`

```
7 > 4
#> [1] TRUE
9 < 3
#> [1] FALSE
```



Recall the example in [The Assignment Operators](#) where we accidentally included a space in `<-` and produced a comparison. Be very careful with spaces. Whitespace can (and should) be included around the comparison operators to help make the code clear. It's all too easy to miss the difference between `x < -3` and `x <-3`.

We can also allow for the possibility of "is x greater than or equal to y?" (`x >= y`) and its partner "is x less than or equal to y?" (`x <= y`)

```
5 >= 6
#> [1] FALSE

3 <= 3
#> [1] TRUE
```

We can test if two numbers are equal to each other with a 'double equals' (`==`) which asks "is x the same value as y?" (`x == y`). The opposite question can be asked also; "is x a different value to y?" or "does x not equal y?" (`x != y`)

```
3 == 3
#> [1] TRUE

7 != 4
#> [1] TRUE
```

Some care needs to be taken with these operators. They test whether or not two values appear to be the same, but not whether they are *precisely* the same. Even though we can specify an integer-like value as either an integer or a real number, these are stored differently. They represent the same value though, so `==` treats these as equal when testing

```
5L == 5
#> [1] TRUE
```

If we truly want to test whether these are the same thing, the `identical()` function checks the types of the input values before declaring things identical

```
identical(x = 5L, y = 5)
#> [1] FALSE
```



### Comparisons between real values

Be very careful when comparing non-integer (real) numbers against each other. Recall from [Getting to Know R Data Types](#) that computers store data in binary (as 1s and 0s). Because of this, there is a limitation to the precision in which decimal numbers can be stored. You may enter the value `0.3`, but the computer attempts to store as many digits (zeroes in this case) at the end of that as it can. In the same way that we can't really write  $1/3 = 0.33333\dots$  exactly without going on forever, computers can't store values that aren't exact powers of 2 without being off by a little.

We can see this in effect by requesting more digits in the output from a `print()` command

```
print(x = 0.3, digits = 17)
#> [1] 0.2999999999999999
```

**When mathematical operations take place, these extra or lacking bits contribute also, so while we may see a nice rounded value in our output, in terms of the digits the computer knows**

about it may be slightly off from that, leading to unexpected results

```
print(x = 0.1 + 0.2)
#> [1] 0.3

print(x = 0.1 + 0.2, digits = 17)
#> [1] 0.30000000000000004
```

This isn't unique to R.<sup>27</sup> rather it's inherent in any computer language. The safest way to avoid this issue; don't compare real numbers against each other if you need the answer to be exact

```
0.1 + 0.2 > 0.3
#> [1] TRUE
```

The `!` in `!=` represents the notion of 'not', as in 'not equal'. This can actually appear in several different places with the same effect

```
3 != 4
#> [1] TRUE

!(3 == 4)
#> [1] TRUE

(! 3 == 4)
#> [1] TRUE
```

What if we try to compare to a missing value? The comparison of anything to the missing value `NA` is simply `NA`.

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

7 == NA
#> [1] NA
```

even if the comparison involves a 'not' operator

```
5 != NA
#> [1] NA
```

Comparison between the two logical values can be summarised in a 'truth table' where the intersection of a row and a column shows the result of the comparison `row == column`, as shown in Table 3.1. Notice that comparing anything to `NA` yields `NA`, including `NA` itself.

**Table 3.1. Truth table for the equals operator (`==`)**

<code>==</code>	<code>TRUE</code>	<code>FALSE</code>	<code>NA</code>
<code>TRUE</code>	<code>TRUE</code>	<code>FALSE</code>	<code>NA</code>
<code>FALSE</code>	<code>FALSE</code>	<code>TRUE</code>	<code>NA</code>
<code>NA</code>	<code>NA</code>	<code>NA</code>	<code>NA</code>

Comparing anything to `NULL` results in nothing, but a particular 'type' of nothing — still a logical value, but one that is of length 0

---

<sup>27</sup> <http://0.30000000000000004.com/>

```
3 > NULL
#> Logical(0)

TRUE == NULL
#> Logical(0)

NA != NULL
#> Logical(0)
```

There are other ways to compare logical values; we can also combine these with *and* (&) and *or* (|) operators. We can make another truth table for the logical options, as shown in Table 3.2 for *and*.

**Table 3.2. Truth table for the and operator (&)**

&	TRUE	FALSE	NA
TRUE	TRUE	FALSE	NA
FALSE	FALSE	FALSE	FALSE
NA	NA	FALSE	NA

The surprising result of Table 3.2 is the combination of NA and FALSE which yields FALSE. The help page for ?`&` explains:



#### From ?`&`

NA is a valid logical object. Where a component of x or y is NA, the result will be NA if the outcome is ambiguous.

A similar table can be constructed for the *or* (|) operator, as show in Table 3. 3.

**Table 3.3. Truth table for the or operator (|)**

	TRUE	FALSE	NA
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NA
NA	TRUE	NA	NA

Again we see that comparison with NA doesn't necessarily lead to NA as long as the comparison is unambiguous.



#### Individual comparisons

While it may be tempting to think that you can write logical combinations the way you might read them, it's important to remember that only one comparison can be made at a time. If we wanted to test if some variable x was greater than 3, we would write

```
x <- 4
x > 3
#> [1] TRUE
```

If we also wanted to test if x was less than 5, one might try simply adding in this condition as 'also less than 5'

```
x > 3 & < 5
```



Error: unexpected '<' in "x > 3 & <"

This fails because when R reads in this expression it breaks it down into the appropriate levels of precedence (recall [Operator Precedence](#)) for which `<` has a higher precedence (is evaluated earlier) than `&`, but here `<` has nothing to compare to, so the call fails.

Instead, we need to repeat ourselves a little and make the two comparisons explicit

```
x > 3 & x < 5 ①
#> [1] TRUE
```

① equivalent to `(x > 3) & (x < 5)`.

There are also double forms of *and* (`&&`) and *not* (`||`) operators which evaluate left-to-right one element at a time until a result is definitely reached. This becomes important and useful when we have collections of logical comparisons we wish to test, but only require a single answer. We may need to make a thousand `&` comparisons to test if all of the values of two inputs are the same, but if they differ in the fourth pair then they're *definitely* not the same, so the rest of the comparisons can be skipped.



#### Equal to NA?

If we need to compare whether or not a value is NA there is a handy built-in function `is.na()` which returns `TRUE` if the input is NA, and `FALSE` otherwise

```
is.na(x = 7)
#> [1] FALSE

is.na(x = NA)
#> [1] TRUE
```

## 3.5 Automatic Conversion (Coercion)

Occasionally, R will perform a conversion on your behalf (whether you wanted to or not). Typically, this is a useful feature; if you wish to add an integer and a non-integer, you might try

```
3L + 2.5
#> [1] 5.5
```

The result of this can be seen by examining the structure

```
str(object = 3L + 2.5)
#> num 5.5
```

so, what has happened here? The C code underlying the `+` function has a few lines specifically for this scenario. If **one** (and only one) of the arguments (with the shorthand `+` notation, the left or right side) is of type `numeric` and the other is of type `integer` then the integer value is *coerced* to a numeric value, then the two are added, producing a final numeric value. This is the result you want — converting an integer to a numeric value is essentially trivial, no rounding is required. Converting

from a numeric value to an integer however has some potential complications, as we saw in [Specifying the Data Type](#). If automatic coercion takes place, R will always coerce to the more general structure. The ordering of these from most specific to most general is

**logical → integer → numeric → character**

R considers certain values to be equivalent to TRUE and other equivalent to FALSE. For numbers, the value 0 (or 0L) converts to FALSE, while any other (positive or negative) number converts to TRUE. We can observe this by requesting the conversion explicitly

```
as.logical(x = 0)
#> [1] FALSE

as.logical(x = 1)
#> [1] TRUE
```

or by comparing with the approximate test ==

```
0 == FALSE
#> [1] TRUE

1 == TRUE
#> [1] TRUE
```

The automatic conversion the other way means that we can add logical and real values together

```
8 + TRUE
#> [1] 9
```

This becomes very handy when we wish to count binary values

```
sum(TRUE, FALSE, TRUE, TRUE)
#> [1] 3
```

Performing comparisons is when you are most likely to encounter automatic coercion. You may not expect the following to work at all, but it does

```
"a" > 5
#> [1] TRUE
```

What's happening here is when R notices you're trying to compare two different types, it performs coercion to the most general type, in this case `character` is more general (it's the most general) than `numeric`. What happens next actually depends on where you live; the strings/characters are compared by their place in the encoding scheme's table. This is language-dependent (which tends to depend on the 'locale' your computer recognises), but this example is using (Australian) English with a UTF-8 scheme (`en_AU.UTF-8`), for which letters come *after* numbers. The `help()` page for the comparison operators (e.g. ?>) notes that in Estonian, "Z" comes between "S" and "T",<sup>28</sup> so beware when using this feature that your results may differ from someone

<sup>28</sup> [https://en.wikipedia.org/wiki/Estonian\\_orthography](https://en.wikipedia.org/wiki/Estonian_orthography)

else's.

This is also the reason that the wayward space in our assignment operation back in SSassignment produced a result; recall that we tried to assign the value 3 to the variable a but accidentally inserted a space inside <- , and so instead generated a logical value since a already held the value "x"

```
a <- "x"
a <- 3
#> [1] FALSE
```

Here R sees a comparison between two different types, so R converts to the most general type (in this case character) and checks the encoding scheme's table to see if "x" comes before or after -3. In my locale, letters come *after* numbers, so this comparison returns FALSE.

We could reasonably expect that the following is automatically coercing to numeric

```
"2" < "3"
#> [1] TRUE
```

which would produce the output as shown. A simple change shows that this isn't the case

```
"2" < "13"
#> [1] FALSE
```

In these cases, no coercion is required as both values are already of type `character`. They are again compared via their positions in the encoding table, and these are sorted numerically such that "13"(starting with a "1") is before "2".

## 3.6 Try It Yourself

 Daily temperatures are typically found in one of two scales; Celsius and Fahrenheit. It's useful to know how to convert between the two. If we have a temperature in Fahrenheit and we wish to convert it to Celsius, we can enter the Fahrenheit value into this expression to find the Celsius value



```
Celsius <- (5 * Fahrenheit - 32) / 9
```

So, if we have

```
temp_F <- 88
```

we can calculate the temperature in Celsius using

```
temp_C <- 5L * (temp_F - 32L) / 9L
```

(the L specifications aren't essential, but are good practice). We can now examine what value this takes

```
temp_C
#> [1] 31.11111
```

Going the other way, we can invert the expression



```
Fahrenheit = (9 * Celsius / 5) + 32
```

so we should be able to obtain our original value again

```
temp_F_recalculated <- (9L * temp_C / 5L) + 32L
temp_F_recalculated
#> [1] 88
```

We can reassure ourselves that this is in fact the same value

```
temp_F == temp_F_recalculated
#> [1] TRUE
```

Evaluate the above conversions (or better yet, write your own) and convert 0°C to Fahrenheit. Convert 100°F to Celsius. Convert -40° from one scale to the other.

## 3.7 Summary

In this chapter we've combined values by adding, subtracting, multiplying, joining, and comparing. We've also seen what missing data looks like and how it interacts with those combination operations.

You've learned that:

- You can use R as a calculator more or less as you'd expect
- Certain types of data can't be added/subtracted
- Operators have different precedence, which can be overridden with parentheses
- Strings can be combined using `paste()`
- Coercion will result in data being converted to the most general type
- Character is the most general type of data
- Text can be compared to numbers because the latter will be coerced to character
- Comparisons between real numbers is dangerous due to rounding differences
- Comparing `NA` to anything produces `NA`, even `NA` itself

New terms you've learned:

### **operator**

a symbol representing an operation to be performed on one or more values, e.g. `+`.

### **precedence**

the order in which operations will be evaluated.

### **expression**

a command for R to interpret; performing an operation on 0 or more data values.

Things to remember:

- If needed, R will coerce data to a common type. When it does, it will coerce to the most general type available.
- Comparing real values is fraught with danger and should be avoided.
- Comparing anything to the missing value NA results in NA.
- Strings can be compared, but doing so depends on your computer's settings, usually dependent on where you live.

We've performed various operations on various types of data, but we've had to know in advance what those values were. Now it's time to learn how to generalise those calculations so they can be performed for **any** input, using *functions*.

# *Understanding the Tools We'll Use — Functions*

## **This chapter covers:**

- How to repeat operations with functions
- The boundaries of where things are defined (scope)
- Messages, Warnings, and Errors, and how to deal with them

The true power of a programming language arises when we can tell the computer to do something, and that request is simpler than just doing the work ourselves. Any time we wish to calculate something more than once it's potentially a good idea to wrap the code up into a meaningful expression that can be re-evaluated with a different (or even the same) input.

We'll quickly see that *functions* are the standard method of achieving this goal in R. We'll also see how to manage things not going to plan once the inevitable errors arise.

## **4.1 Functions**

A *function* (in programming) refers to a series of defined operations/a routine which achieves some task, usually involving data, producing some result. Functions can have input data (or none) and produce output data (or none) or perform some action. Functions may also define or use more data if it is defined by the function itself (e.g. constant numbers or values).

In R, functions are handled in the same way that data are — stored with a variable name, and can be sent as input to other functions. Saving the operations in a named function (the *body* of a function) has many advantages, including

abstraction	the function can be provided with different data.
re-usability	reducing copy-and-paste repetition of the same operations which can lead to errors.
resilience	checks can be performed on the input data to ensure it meets the assumptions of the operations to follow.

R is a *functional* language;<sup>29</sup> it uses *functions* to interact with data, returning some other output and otherwise leaving the input unchanged (as opposed to a *procedural* language, which uses *statements* to modify data with an output dependent on the current state of the system). In a *functional* language, data is never really changed, and this holds in R.<sup>30</sup>

If we wish to repeat an operation, perhaps generalise it to arbitrary inputs, we can do this using a *function*. If we frequently needed to calculate the area of a circle, could do this ourselves each time with the right formula (a similar enough concept to a function). The formula for the area of a circle is

$$A = \pi r^2$$

where  $\pi = 3.14159\dots$  is a built-in constant in R named `pi`. To calculate an area  $A$ , all we need to know is the radius  $r$ , which we square (multiply by itself) and then multiply by `pi`. We could do this several times ourselves for radius values of 3, 4, and 5. Recall that we can raise a value to a power with the `^` operator, so to calculate the area ( $A$ ) of circles with these radii ( $r$ ) we calculate  $\pi$  multiplied by the radius ( $r$ ) to the power of 2

```
pi * (3 ^ 2)
#> [1] 28.27433

pi * (4 ^ 2)
#> [1] 50.26548

pi * (5 ^ 2)
#> [1] 78.53982
```

or we could create a function that does this for us if we supply the radius. Functions are stored as objects in R, so we name these using the assignment operator like any other object/variable.

While it has been straightforward enough to enter simple commands into the `Console`, more complex commands become a bit too unwieldy to enter this way. Instead, it is usually better to work with a *script* in the `Editor` pane.

 Create a new script file using the `File → New File → R Script`. Enter the following code into the `Editor` pane

```
areaCircle <- function(radius) {
  1 2 3
  area <- pi * radius ^ 2
  4
  return(area)
  5}
```

- 1 The keyword `function()` tells R that we are defining a function.
- 2 We define this to be a *function* of the *input* `radius` which will be allowed to vary.
- 3 The *body* of the function is bounded by `{` and `}`.
- 4 We can create (temporary) variables inside functions; we'll see why these are temporary in [Scope](#).

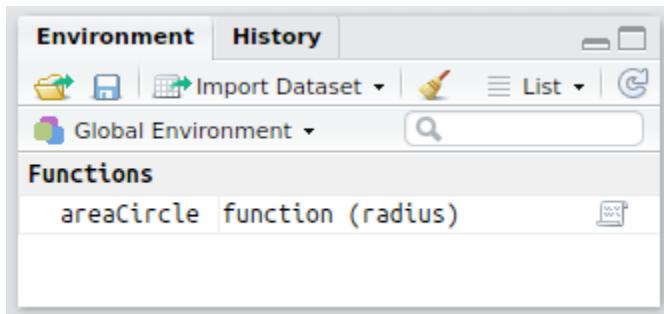
<sup>29</sup> An *impure* one, in that not all functions provide *referential transparency* (where an expression could be replaced by its output value) and many functions produce *side-effects* (such as plotting).

<sup>30</sup> For the most part. The `data.table` package bypasses this restriction by acting on data in-memory, which greatly increases the speed of its operations.

- ⑤ A function *returns* (outputs) a value which is the result of the calculation.

Highlighting the function definition and evaluating it (by pressing **Ctrl** + **Enter** or **Cmd** + **Enter**) makes the function available for use, and it now appears in the Environment listing as shown in Figure 4.1.

**Figure 4.1. Function defined**



We can now calculate the area of a circle using only the minimal information (a radius value) and our function. When we evaluate this function with a value as the radius argument value, R performs the operations stored in the *body* of the function, and then *returns* (outputs) the value identified by the `return()` call. By default, outputs are printed, so we see the results of our calculations printed to the Console

```
areaCircle(radius = 3)
#> [1] 28.27433
areaCircle(radius = 4)
#> [1] 50.26548
areaCircle(radius = 5)
#> [1] 78.53982
```

In this way, we can *abstract* away calculations and name them appropriately; we don't need to explicitly write out the area formula each time we wish to use it; we now have a shortcut to that in the form of our function. In the above example, we named our function according to what it calculates (the area of a circle); you'll likely be calculating many different things, so take some care in how you name your functions.

We have now calculated these areas, but where are the values? You'll notice that nothing more has appeared in the Environment listing. Because of the way that R is designed (*functional*, leaving data unchanged) evaluating a function doesn't change **any** data, so no new data is created in any permanent sense. It's printed, then it's gone.<sup>31</sup> In order to retain the value calculated, we need to assign it to another variable.

---

<sup>31</sup> With the exception that it can still be recalled immediately using `.Last.value` as discussed in [Naming Data \(Variables\)](#).



### On assigning the returned value

A common complaint from people having trouble with their code is that "my function doesn't do anything". These complaints typically involve function calls on data that look like

```
my_data <- 7
my_data
#> [1] 7
increase_my_data <- function(data_input) {
  updated_data <- data_input + 1
  return(updated_data)
}

increase_my_data(data_input = my_data)
#> [1] 8
my_data # (not changed)
#> [1] 7
```

Can you spot what is missing? `increase_my_data` should return something very different from its input, and it does, but this person hasn't assigned that returned value to anything. R evaluates `increase_my_data(my_data)` and prints the value to the Console (by default), but doesn't update the variable `my_data`; that would be a *procedural* way of doing things, whereas R works in a *functional* manner, leaving data alone.

The 'solution' in this case is usually to assign (recall: [Assignment](#)) the output (return value) of `increase_my_data`, in this case back to the original data name:

```
my_data <- increase_my_data(data_input = my_data)
my_data
#> [1] 8
```

Keep in mind that this overwrites the value `my_data`, which may not always be the result you are trying to achieve.

We've seen how a simple function is written and used, but what's actually happening? Building a proper understanding of how R sees functions can greatly improve your ability to use these valuable tools.

#### 4.1.1 Under the Hood

You may write the expression

```
x <- 2
x <- cos(x = 2*x)
```

and think that the object `x` is being updated from some value (2) to the cosine of twice that value,<sup>32</sup> but there's a lot more going on behind the scenes here. In reality, the value `2*x` is provided to the `cos()` function as an input. The `cos()` function is some series of operations to perform involving this input, resulting in a *return value*. This value is then assigned to the variable `x`, overwriting the previous value. Right up until that last step though, the value of `x` was completely untouched. We can confirm this by

---

<sup>32</sup> from trigonometry; together, the sine (`sin()`), cosine (`cos()`), and tangent (`tan()`) functions can be used to describe, for example, the ratios of sides of a triangle

looking at the value before and after calculating  $\cos(x)$ , but not assigning it back to  $x$ :

```
x <- pi      ❶
x
#> [1] 3.141593
```

❶  $\text{pi}$  is one of the few built-in constants.

```
cos(x = 2*x) ❶
#> [1] 1
```

❶ the cosine of  $2\pi$  should be 1.

We can see that although  $x$  appears in this function call, it still has the value we originally assigned to it

```
x
#> [1] 3.141593
```

An equivalent set of expressions to the  $x <- \cos(x = 2*x)$  example is the following:

```
.y <- cos(x = 2*x)
x <- .y

x
#> [1] 1
```

where  $.y$  becomes inaccessible directly. The reason we can shorten this to just  $x <- \cos(x = 2*x)$  safely is because of the way that R performs these operations. Recall that due to the *REPL* nature of R, once you press **Enter** the underlying code takes over and begins digesting your expression. For our above example, the *functions* are unpacked.

### On R functions

**“ To understand computations in R, two slogans are helpful: Everything that exists is an object. Everything that happens is a function call.”**

-- John Chambers *creator of the S programming language and core member of the R programming language project*

We therefore have a neat and tidy way of asking R to:

- calculate the cosine of
- double a value and
- assign that to that same value’s name,

but under the hood, R treats these three operations as *functions*:



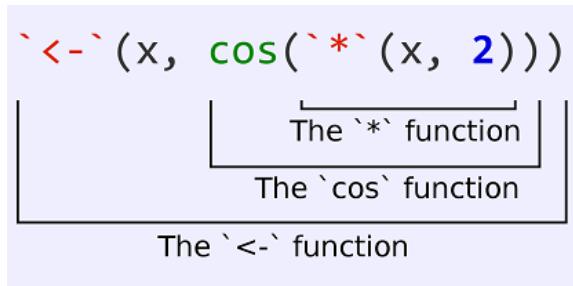
```
cos(a)      ❶
`*`(e1, e2) ❷
`->`(l, r) ❸
```

❶ Input: angle a. Output: cosine of a.

- 2 Input: values e1 and e2. Output: product of e1 and e2, i.e.  $e1 * e2$ .  
 3 Input: expressions l and r. Output: none, with side-effect of assigning the value r to the variable l.

and our tidy little expression is really the *composition* of the three functions, evaluated from the inside outwards: the `\*` function with inputs x and 2 produces a value, which becomes the input to the cos() function, which itself produces a value, which becomes the second input to the `<-` function.

**Figure 4.2. Function composition**



The backticks (`) tell R that we are calling a function directly, so it can skip over trying to unpack this again. The `<-` (assignment) function doesn't return anything (visibly, at least),<sup>33</sup> so nothing is printed to the Console when this expression is evaluated.

We can see now that up until the outermost function, it is the return values of the inner functions that are evaluated in the next outermost function, and x will remain unchanged.

We won't be calling functions in this manner, but it's very useful to keep in the back of your mind that this is what R is really seeing.

### 4.1.2 Function Template

The general structure of a function definition in R is the following:



```
name_of_function <- function(argument1, argument2) {           1 2 3
  temporary_variable1 <- other_function(argument1)             4
  final_variable <- yet_another_function(temporary_variable1)
  return(final_variable)                                       5
}
```

- 1 The function is assigned to a variable just like a value would be. In this case, the variable name is `name_of_function`.  
 2 We'll cover how *arguments* work and how to define them in [Arguments](#), but these are the *inputs* to a function.

<sup>33</sup> Technically, the assignment operator returns a value *invisibly* (not printed to the Console), so it's still there even though you don't see it

- ③ Placing the function body-enclosing brackets on this line or the next is a matter of style. They can even be left off entirely if a function body contains only a single expression.
- ④ Operations are performed on the inputs and potentially other data with more functions and assigned to temporary variables (which functions are possible, and why these are temporary variables will be covered in [Scope](#)).
- ⑤ The *output* of the function, the value that will be returned by calling the function, is identified by a special function called `return()`. Alternatively, R will assume that the last expression in a function produces the value to return.

We *call* a function (possibly with *arguments*) by adding parentheses surrounding any arguments (or no arguments) to the name of the function, so the above template can be called using



```
name_of_function(argument1 = x, argument2 = y)
```

Recall from [The Assignment Operators](#) that the value(s) given to *arguments* is/are specified with `=` rather than `<-`. If we had another function which took no arguments (or they had defaults) then we could simply omit any arguments, but the parentheses remain



```
anotherFunction()
```

Functions can be used whenever you might want to repeat some code, you want to group or isolate some steps of your analysis (perhaps to make it clearer what the purpose of those steps is), or you want to manipulate your data but are only interested in preserving the final result, not the intermediate transformations you create along the way.



### Autoprinting of `return()` values

If no `return()` function is explicitly stated at the end of a function, the result of the last expression within the function is used as the return value. If the result of evaluating a function is not assigned to anything (the function is simply evaluated, e.g. `myFunction(7)`) then typically this will result in the return value being printed in the same way as evaluating a variable would. A special case is when the last expression in a function itself returns no value (or does so invisibly) such as the assignment operator `<-`. In these cases, evaluating the function without assigning the result will produce no output (but the return value can still be assigned).

For example:

```
returningFunction <- function() {
  2 + 7
}
returningFunction()
#> [1] 9
```

prints the value to the Console, while

```
nonReturningFunction <- function() {
```

```

x <- 2 + 7
}
nonReturningFunction()

```

**does not. These two are equivalent however if we assign the results**

```

a <- returningFunction()
b <- nonReturningFunction()
a == b # are these equal?
#> [1] TRUE

```

**In either case, using return() explicitly is highly recommended, especially as this allows you to return() a value from the middle of a function, rather than the end.**

```

bigOrSmall <- function(x) {
  if (x > 10) { ①
    return("x is big") ②
  }
  return("x is small") ③
}

```

① We'll cover what if() does in a later chapter, but perhaps you can already guess.

② Here we are exiting the function from the middle, when x is greater than 10.

③ Here we are exiting the function normally, at the end, when x is not greater than 10.

**The function returns one of the two possible results, either returning normally**

```

bigOrSmall(x = 8)
#> [1] "x is small"

```

**or at another designated point**

```

bigOrSmall(x = 12)
#> [1] "x is big"

```

Since in R "everything that happens is a function call", even seemingly simple things like the addition operation + are of course functions behind the scenes, just cleverly written. The + operator is actually just shorthand for a function that takes two arguments; the code behind R recognises when this is being used and performs the translation for you. We can see this by evaluating + without the parentheses, which returns the function body itself. Enter the following into the Console and press **Enter**

```

`+`
#> function (e1, e2) .Primitive("+")

```

so 2 + 2 is actually the function call (backtick escaped)

```

`+`(2, 2)
#> [1] 4

```

### Just a simple function?

 **Don't let the apparent brevity of function (e1, e2) .Primitive("+) fool you. Behind the scenes is a whopping 116 lines of C code (which calls further functions too) which handles all the different possible things you might mean when you place a + between two other things. R nicely abstracts all of that away so you only need to write the +, whether you're adding a numeric type to an integer type, or one of the values is the missing value NA.**

It can even handle the case when you don't provide *anything* as the first argument, in which case the underlying code ensures that the expression is understood as a *unary* operation (single argument, as opposed to a *binary* operation with two) and gives you back the original value. Nothing interesting there, but the same code handles the arithmetic for the `-` function, in which case the *unary* operation returns the value with the opposite sign.

We haven't covered more complicated structures yet, but this same chunk of code handles those too.

The vast majority of R functions are vulnerable to being overwritten by a user in their session, so once again naming things becomes a challenge. There's nothing stopping you from deciding to name your function `+` and deciding that it should do the same thing as the already defined `-` function

```
`+` <- ` -`
```

This of course wreaks havoc with sensibilities and anything you may want to do involving numbers down the track

```
2 + 2
#> [1] 0
```

so clearly this was a poor choice of function name. If you do this temporarily, it's a very good idea to return things to their normal working order

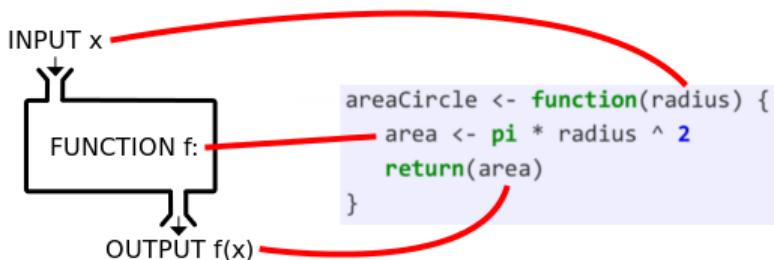
```
`+` <- base::` +`
```

Every operation we perform will be a combination of one or more functions. If we want those functions to know about our data, we need a way to communicate our values. We do that via function *arguments*.

### 4.1.3 Arguments

Now that we know what a function looks like, how do we give a function some data to work with? Here it may be helpful to think of a function like a mechanical processing machine; it takes some input (the raw materials), performs some operations, then outputs something, as shown in Figure 4.3.

**Figure 4.3. The function machine (adapted from Wvbailey [Public domain], via Wikimedia Commons).**



The input could vary, and depending on the internals of the machine, so may the

output. The way that R handles this is to create a new variable for each input that is specific to the function. These are just like the other variables we have been working with, except that (as we will see in [Scope](#)) they exist only in their own little world within the function.

The value of this is *abstraction*; the argument variable now represents some unknown input data that the function needs to work with, and that input data could be different each time the function is evaluated. This concept is present in spreadsheet programs when using formulas; the *arguments* there are usually named in an abstracted way also, such as in figure 4.1.

**Figure 4.4. Arguments to the 'SUM()' function in a spreadsheet, named 'number 1' and 'number 2'.**

	A	B	C	
1	4			
2	8			
3	=sum()			
4		SUM(▶ number 1, number 2, ... )		
5				

Let's make our machine analogy a little more specific, and imagine that our machine takes a word, performs some operations on it, and returns the first letter.

▀ A function that performs this operation might look like the following. Enter this into a *script* in the `Editor` pane of RStudio and evaluate it

```
first_letter <- function(word) {
  ## extract the first letter of the word using substring()
  ## which is called as substring(text, first, last)
  ## see ?substring
  first <- substring(word, 1, 1)

  ## write a message to the Console
  message_text <- paste(word, "starts with", first)
  print(message_text)

  ## return the first letter
  return(first)
}
```

Here we have created a function called `first_letter` which takes one *argument* — a word. This is the *input (argument)* to the function. We can evaluate this function with any *input* value

```
first_letter(word = "cat")
#> [1] "cat starts with c"
#> [1] "c"
```

The first output (beginning with [1]) is the output from the `print()` function. The next output is the value returned from `first_letter()`. The internally created variables `first` and `message_text` aren't available outside of this function; they only exist while the function is being evaluated. So, the following generates an error

```
message_text
```



Error: object 'message\_text' not found

Similarly, we haven't created a variable `word` outside of the function, even though we used `an =` to provide the value to the argument

```
word
```



Error: object 'word' not found

We also haven't assigned the return value to anything, so evaluating the function merely prints the result. We could however use our function to create a new variable by assigning the result

```
first_of_apple <- first_letter(word = "apple")
#> [1] "apple starts with a"
```

Note that the message was printed during evaluation, but the result was not printed. We can retrieve the value by printing it

```
first_of_apple
#> [1] "a"
```

A function is only useful if you need to use it more than once, otherwise you could just write its contents out and be done. If we wished to process several words and save their first letters to new variables, we could do that with a little repetition

```
first_of_cat <- first_letter(word = "cat")
#> [1] "cat starts with c"

first_of_dog <- first_letter(word = "dog")
#> [1] "dog starts with d"

first_of_fish <- first_letter(word = "fish")
#> [1] "fish starts with f"
```

This function was only short, but hopefully you can appreciate that abstracting longer series of operations has great value. As it stands, we have made our function fairly general — it is able to extract the first letter of a word and we don't need to write out all of the steps to do so each time.

#### 4.1.4 Multiple Arguments

We're not limited to just a single argument though; we can have as many as we require. The processing we do between input and output may involve several named or unnamed arguments. There are a couple of ways to do this; we could either pass in a

more complex object as the input (we'll get to how to do this later), or we could pass in several arguments. This is the safer option, because it's easier to track what the inputs are.

Functions aren't limited to inputs of strings either; we could have numbers, dates, any type of R object in fact, even more functions. This is a powerful feature of R that you'll be thankful you learned.

If our function knows what to do with more arguments, it will use them. If it receives an argument that it doesn't know how to process, it will produce an error

```
first_letter(word = "banana", music = "jazz")
```



Error: unused argument (music = "jazz")

This gets somewhat trickier when functions use the 'dot-dot-dot' syntax (...) which can process **any** arguments, so just be extra careful in that case.

The opposite is also worth mentioning; just because a function *has* some arguments doesn't mean that you absolutely have to use them. This is highly dependent on the function (particularly whether or not it really *needs* certain arguments). We could create a trivial function that takes an argument, does absolutely nothing with it, and returns the number 3

```
trivial_function <- function(arbitrary_argument) {
  return(3)
}
```

Calling this with or without the argument works just fine

```
trivial_function(arbitrary_argument = 7)
#> [1] 3
trivial_function()
#> [1] 3
```

In fact, R will stop evaluating arguments at all as soon as it can. This is known as *lazy evaluation* and it both helps speed up code and makes it more flexible. If we added another argument that wasn't required and wasn't used, say

```
lazy_function <- function(x, y) {
  return(x)
}
```

then R doesn't even mind if we try to use a variable that doesn't exist in the second argument, since it never attempts to evaluate it

```
lazy_function(x = 2, y = undefined_variable) ①
#> [1] 2
```

① `undefined_variable` has not been defined, but R won't even try to evaluate it.

If we expand our function to include some processing of the output, we can pass in a function name as well

```
first_letter_formatted <- function(word, fun) { ①

  ## extract the first letter of the word using substring
  first <- substring(word, 1, 1)

  ## write a message to the Console
  message_text <- paste(word, "starts with", first)
  print(message_text)

  ## return the first letter
  return(fun(first)) ②

}
```

- ① Note now that we also have the argument `fun`
- ② We return the result of `fun(first)`

Take some time to process what we're intending to happen here; we have created a function that will call some other function (which we haven't defined). If this seems a little haphazard and lacking of rigor, well, that's understandable because we haven't added any guarantees about the function `fun`, what arguments it needs, or what it will return. Let's start with our own function. What if we just try to evaluate it without the second argument?

```
first_letter_formatted(word = "dog")
#> [1] "dog starts with d"
```



Error: argument "fun" is missing, with no default

We can see that the function `message` was created and sent to the Console, but the rest failed when trying to evaluate `fun(first)`. That makes sense; we haven't told our function what `fun` should be.

We need a function which takes a string, does some formatting, and returns another string. There's nothing that says this in our code, but that's what we're intending. We could make that clearer by adding some comments to that effect



Good documentation makes it clear how your function **should** be used; it defines a **contract** between you, the author/user, and the function. You agree to pass in certain types of data, and the function agrees to process them in a certain way, returning a certain type of result.

Use comments to describe what the inputs and outputs **should** be, and what you intend the internal processing to accomplish. For example, a trivial example might be

```
## adds one to a number
## input: n, a number
## output: a number, n+1
add_one <- function(n) {
  return(n + 1)
}
```



There are much more sophisticated ways to do this once you start writing packages, including testing (`testthat`) of these contracts and automatic documentation of arguments (`roxygen`).

Some functions we could use are the `toupper` and `tolower` functions which convert strings to uppercase or lowercase, e.g.

```
toupper(x = "text processing is fun")
#> [1] "TEXT PROCESSING IS FUN"
```

We can pass one of these to our function

```
first_letter_formatted(word = "dog", fun = toupper)
#> [1] "dog starts with d"
#> [1] "D"
```

Note that we didn't pass `toupper` as a string, and we also didn't *call* the function (which we would do with parentheses, i.e. `toupper()`). We passed the actual function to `first_letter_formatted` so that it was used internally as if we had defined it there. We can look at what we passed in (the function itself) by evaluating it, again not including the parentheses (which would mean we are *calling* the function)

```
toupper ①
#> function (x)
#> {
#>   if (!is.character(x))
#>     x <- as.character(x)
#>   .Internal(toupper(x))
#> }
#> <bytecode: 0x2aeadb0>
#> <environment: namespace:base>
```

- ① Note here we haven't used the parentheses; we're asking for the function *code*, not *calling* the function.

We can pass in a different function easily

```
first_letter_formatted(word = "DOG", fun = tolower)
#> [1] "DOG starts with D"
#> [1] "d"
```

In fact, our `first_letter_formatted` function can take any compatible function as the `fun` argument, as long as it can process a string when called as `fun(x)`. This flexibility makes R amazingly powerful as it can build incredibly complex function out of smaller components, passing around functions and data as required.

## 4.1.5 Default Arguments

In our previous example we extracted the first letter of several words. If we extend those to include our formatting, we begin to recognise some repeated code

```
first_of_cat <- first_letter_formatted(word = "cat", fun = toupper)
#> [1] "cat starts with c"
first_of_dog <- first_letter_formatted(word = "dog", fun = toupper)
#> [1] "dog starts with d"
first_of_fish <- first_letter_formatted(word = "fish", fun = toupper)
#> [1] "fish starts with f"
```

There's an old adage in programming known as 'DRY' (Don't Repeat Yourself) which encourages removing unnecessary repeated structures in favor of functions or better design. In this case, we may wish to use `toupper` most of the time, and we can set this as the *default* argument. In this case, the argument `fun` will still be recognised if requested, but will take a particular value by default.

The way to instruct R that we wish to use a default argument value is to specify it when defining the function, so we would have

```
first_letter_formatted <- function(word, fun = toupper) { ❶
  ## extract the first letter of the word using substring
  first <- substring(word, 1, 1)

  ## write a message to the Console
  message_text <- paste(word, "starts with", first)
  print(message_text)

  ## return the first letter
  return(fun(first))
}
```

- ❶ If this function is called without the argument `fun`, it is given the value `toupper`.

Now we can use this function, with the formatting behaviour, despite only providing one argument

```
first_letter_formatted(word = "house")
#> [1] "house starts with h"
#> [1] "H"
```

If for any reason we don't want to use the default, we can specify the value we do want to use

```
first_letter_formatted(word = "HOUSE", fun = tolower)
#> [1] "HOUSE starts with H"
#> [1] "h"
```

You've actually seen this in action already — many R functions have default values so you don't need to specify them, one of those being `print()` which we have used to print an object to the Console.

```
print(x = "a")
#> [1] "a"
```

If, instead, we wanted to leave the quotes off, we could use the `quote` argument (which has a default value of `TRUE` for strings)

```
print(x = "a", quote = FALSE)
#> [1] a
```

If you are using RStudio, pressing `Tab` after typing a comma when evaluating a function should bring up a tooltip window of the possible arguments to that function. The definitions for each of these (and whether or not they have default values) are usually found in the help file for that function.<sup>34</sup>



#### On unwanted defaults

**Be mindful of default argument values; they aren't always what you might expect them to be.** A particularly painful one for many R users appears when importing data in which case the `default stringsAsFactors = TRUE` performs conversion on any text data automatically, and usually contrary to what is desired.

### 4.1.6 Argument Name Matching

Until now I've tried to make sure that I explicitly name input variables in function calls, such as

```
print(x = "abc")
#> [1] "abc"
```

but do I really need to use `x =` every time? For better or worse, the answer is "no". R will gladly make some assumptions about the input variables you use in function calls. If you don't name any of the inputs, R will assume that they're being provided in the order in which they appear in the function definition. This is known as *positional* indexing. When functions only have a single argument, this becomes trivial, and the name is easily dropped. When a function takes several arguments however, if you aren't going to use their names, it's important to ensure that you input them in the correct order. We can use the `print()` function above just fine without specifying that the argument name is `x`; R will figure that out for us

```
print("abc")
#> [1] "abc"
```

For example, to calculate a random value we can use the `runif()` (random uniform value) function from the (by default, available) `stats` package. If we lookup the help file for this function using `?runif` then we see that it takes three arguments;

- `n`: the number of values to generate.
- `min`: the minimum value that could be produced.
- `max`: the maximum value that could be produced.

This means we can request `n` random values between `min` and `max`. We can keep working with the same 'random' variables consistently if we set the `seed` value using

<sup>34</sup> Recall, that the help file for some function `foo` can be reached either by pressing `F1` while your cursor is on the function name, or by evaluating `?foo`.

the `set.seed()` function before evaluating a random selection, which I've done in each case here so that the results are consistent.<sup>35</sup>

If we are being (correctly) verbose and writing out argument names, we can request four values between 1 and 2 from `runif()` with

```
set.seed(1)
runif(n = 4, min = 1, max = 2)
#> [1] 1.265509 1.372124 1.572853 1.908208
```

If we're confident in these arguments, we can simply drop their names (note, these are random, but I've asked for the *same* random selection, so they are the same values as those we just obtained)

```
set.seed(1)
runif(4, 1, 2)
#> [1] 1.265509 1.372124 1.572853 1.908208
```

Coming across this line with unnamed arguments 4, 1, and 2 in the middle of a large script should raise some eyebrows, especially if the function being called is more obscure. It's difficult to read into what those numbers represent when they are passed by position and not their names.

An added benefit of using the argument names is that they can commonly be provided in any order

```
set.seed(1)
runif(max = 2, n = 4, min = 1)
#> [1] 1.265509 1.372124 1.572853 1.908208
```

The very large caveat to this is when functions utilise the dot-dot-dot ellipses (...). This is a special argument which captures additional arguments and passes them to an internal function call, so one is able to send these arguments 'down the line'

```
nestedFunction <- function(N, ...) {
  runif(n = N, ...)
}
```

In this example, the ... argument will capture any additional arguments other than N (since it has no other named arguments) and use them in the `runif()` call

```
set.seed(1)
nestedFunction(N = 3, min = 1, max = 2)
#> [1] 1.265509 1.372124 1.572853
```

Note that the definition of `nestedFunction` doesn't specify `min` or `max` at all, it just allows for 'other arguments' which it will pass to `runif()`.

If we choose not to use these, then nothing is passed to the `runif()` call and the default values are invoked (`min = 0, max = 1`)

<sup>35</sup> This is good practice any time you are using random numbers. Some people have a favourite seed, e.g. their birthday, 42, or 1337. If you're evaluating all of your code in order, then the seed only needs to be set once. I have set it each time here so you can re-run the examples in any order and get the same results.

```
set.seed(1)
nestedFunction(N = 3)
#> [1] 0.2655087 0.3721239 0.5728534
```

(note that since we're not using the arguments we had previously been using, these aren't the same random variables as we previously obtained). We don't even need to name these additional arguments; the first will be assumed to be the first positional argument, and the remaining arguments will be captured by ...

```
set.seed(1)
nestedFunction(3, 1, 2)
#> [1] 1.265509 1.372124 1.572853
```

The other useful place you'll commonly find this feature is when a function requires an indeterminate number of arguments, such as `sum()` which calculates the sum of its arguments, no matter how many, without requiring a name for each value

```
sum(12, 13, 9, 11)
#> [1] 45
```



#### Similar but different

Be warned though; different functions have different arrangements, names, and defaults for their arguments. While `sum()` takes any number of arguments and calculates the sum of their values, you may expect `mean()` (calculate the average) to do something similar. Of course, I wouldn't be mentioning it if it did.

If we look at the help file for the `sum()` function (using `?sum`) we see that the first argument (and only unnamed argument) is ... which means "any number of variables not specifically named here"

```
sum(1, 2, 3, 4)
#> [1] 10
```

If we try that with the `mean` function, which should take the average of the values, as

```
mean(1, 2, 3, 4)
#> [1] 1
```

something isn't right. The average of those values is 2.5. If we go back to the help files (`?mean`) we see that `mean()` has an `x` argument before the ... and the Value description notes that the function calculates the mean of `x`. In this case, the function is expecting more than one value to be stored in `x`, and we'll see how to do that when we cover combining values into larger structures.

You'll probably want to look at the help file for every function you use, at least once. When written well, the help file contains a useful description of how and when to use a function, potential hazards to watch out for, and links to other similar or related functions. If you're using a function and not getting the answer you expect, the help file is the first place to check.

#### 4.1.7 Partial Matching

There are many things that R does in order to try to help you get your work done. For better or worse (probably 50/50 most of the time) many of these things are quirks of

the language. One of those is the fact that if you supply enough of an argument name that R can figure out which one you mean, then R considers that to be enough.

This behaviour is called *partial matching* of arguments and it can either be a blessing (rarely) or a curse (more often). Let's say you want to generate ten random numbers, drawn from a normal distribution.<sup>36</sup> The function to generate these values is `rnorm()` and is available by default from the `stats` package. It has the arguments

- `n`: the number of values to generate.
- `mean`: the mean of the normal distribution (default 0).
- `sd`: the standard deviation of the normal distribution (default 1).

Calling this function generates random numbers with a higher probability of being near the mean (in this case, the default value of 0). Again we'll use the `set.seed()` function to reproducibly sample randomly. Let's select ten values using `rnorm`

```
set.seed(1)
rnorm(n = 10)
#> [1] -0.6264538 0.1836433 -0.8356286 1.5952808 0.3295078 -0.8204684
#> [7] 0.4874291 0.7383247 0.5757814 -0.3053884
```

The defaults for the arguments `mean` and `sd` are 0 and 1, respectively. If we want to call `rnorm()` with different arguments, we certainly can

```
set.seed(1)
rnorm(n = 10, mean = 10, sd = 2)
#> [1] 8.747092 10.367287 8.328743 13.190562 10.659016 8.359063 10.974858
#> [8] 11.476649 11.151563 9.389223
```

but we can be lazy and only supply enough of the variable names such that R knows which ones we're talking about

```
set.seed(1)
rnorm(n = 10, m = 10, s = 2)
#> [1] 8.747092 10.367287 8.328743 13.190562 10.659016 8.359063 10.974858
#> [8] 11.476649 11.151563 9.389223
```

Note that we didn't specify the `mean` and `sd` arguments, but `m` and `s` partially match to those uniquely, so R goes ahead and works with what it has. This works when the function is defined with named arguments (these are called *formal* arguments) rather than the catch-all ... which doesn't specify what name the arguments must have.

As you can imagine, this isn't always so smooth. Often arguments will have many letters in common at the start of their names, so it's best to not rely on this behaviour at all. For example, if instead of a normal distribution we wanted random numbers drawn from the uniform distribution we saw in the previous section, we could call the `runif()` function. Calling this with fully named arguments works as expected

```
set.seed(1)
runif(n = 10, min = 0, max = 1)
```

<sup>36</sup> Knowing exactly what this means isn't critical, we're just specifying how likely it is to produce each number. In this case we're more likely to produce a number close to the `mean` (average) value.

```
#> [1] 0.26550866 0.37212390 0.57285336 0.90820779 0.20168193 0.89838968
#> [7] 0.94467527 0.66079779 0.62911404 0.06178627
```

but trying to partially match with just a single character fails since it doesn't match uniquely to a single argument

```
set.seed(1)
runif(n = 10, m = 5)
```



**Error: argument 2 matches multiple formal arguments**

It's best practice to always specify your argument names in full so as not to rely on this fragile feature, or to cause confusion with raw data being assigned to arguments by position alone.

#### 4.1.8 Scope

It's important to understand what's happening to variables that are created and manipulated within a function. I've told you that once a variable is used as the value of an argument, it's left alone and unchanged by whatever happens inside the function, but what if we try to create an internal variable with that same name? One easy way to find out of course, is to try!

```
x <- 2                                1
print_x <- function(x = 7) {           2
  print(x = x)                          3 4
  x <- 4                                5
  return(x)                            6
}
print_x(x = x)                         7
```

That's a lot of xs to keep track of! We have:

- ① The variable x that we created with the value 2,
- ② The argument name x to our print\_x() function, with default value 7,
- ③ The argument name x to the print() function,
- ④ The value x provided to the x argument of the print() function,
- ⑤ The x with a value of 4,
- ⑥ The variable x returned from print\_x()
- ⑦ The expression x to be evaluated, which by default, will print() the value.

With trial and error, you might be able to figure out the rules that R will follow, but I'll tell you what they are then you can guess what the final evaluation will produce.

Most programming languages have some concept of *scope* — the parts of code for which a variable (or object in general) will be relevant. Consider for a moment the first names of everyone you know; Alice, Bob, Chris, ... This is the widest *scope* of 'people names' you have. Now, consider you move to another country away from everyone you know. Now you begin to create a new list of 'people names' as you meet new people. This scope is *narrower* because it is more relevant to you right now, in your new home. Some of these new people might have the same first name as those you knew back home, but the two Bobs are still different people, they just have the same label.

In this same way, while R is evaluating the instructions within a function, it creates new variables which may or may not have the same name as variables you defined outside of the function, but internally the memory addresses (or last names, if you wish to continue the analogy) of these are different. So, the internal variable `x` inside our `print_x()` function might have the same name as `x`, but it's a new variable specific to that function. If it helps, think of this as `x_2`; the label `x` is only for our understanding, R knows what it's doing. Variables defined within a function's scope are commonly referred to as *local* variables. We still use the assignment operator `<-` to store these, but they are stored (and exist) only within the scope of the function.

What happens to these variables once the the function has completed? This is where the analogy to people in different countries breaks down. While 'local' Bob will probably live on after you return home, the local variables created inside a function do not. This is part of the *functional programming* paradigm; data should be unchanged, and functions shouldn't have *side effects* which means that merely evaluating a function shouldn't change any data.

Can you guess what the code above will produce yet? We now know that the value `x` won't be changed by the function, even though it temporarily creates another variable named `x`. Let's run the code and see. First we assign the value 2 to `x`

```
x <- 2
```

Now we define the function

```
print_x <- function(x = 7) {
  print(x = x)
  x <- 4
  return(x)
}
```

If we evaluate the function with our variable `x` given to the argument `x`, we see

```
print_x(x = x)
#> [1] 2
#> [1] 4
```

The first result comes from the `print()` function, which prints the value of the local variable `x`. Since we don't assign the return value of the function to anything, it is also printed to the `Console`. If we now check what value `x` has

```
x
#> [1] 2
```

we see that it remains unchanged.

What if we evaluate just the new function without any arguments?

```
print_x()
#> [1] 7
#> [1] 4
```

In this case, since we didn't specify what the local version of `x` should be, it is assigned the default value, which is what the `print()` function receives. This is printed, then this local variable is assigned the value 4, which is returned and thus printed. Our original variable `x` however is still unchanged

```
x
#> [1] 2
```

Of course, a very useful solution to this is to name your variables appropriately. `x` is almost always a bad choice of name for data, but can make sense as an argument to a function if it is a completely general input.

The local variables are essentially destroyed once the function evaluation is completed, which means we can't access them outside of that *scope*. If we try to, R will complain and return an error

```
y_in_scope <- function(y) {
  print(x = y)
}

y_in_scope(y = 5)
#> [1] 5
```

Here the value of `y` only exists within the *scope* of the function, so we can't use it outside of that

```
y
```



**Error: object 'y' not found**

This can actually be of great benefit — if your function requires several intermediate steps then the intermediate values only exist for as long as the function is being evaluated. The inputs and outputs are all that remain once the function evaluation is completed, so we don't have unnecessary variables floating around.

R will search through the different levels of *scope* when it tries to find what value a variable might have (if it has one). It begins with the scope which the expression is in (a function, or the workspace). If R can't find a variable with that name in that scope, it moves to wider and wider scopes. With functions inside functions, R moves up through the order in which they were called until it reaches the workspace. If it fails to find the variable there, it searches the loaded and attached packages (see [Packages](#)). Only when it fails to find the variable in any of these does it return the error.

This can be very useful, but also very dangerous. In our example directly above we

showed that the variable `y` would not persist outside of its function's scope. The opposite scenario however occurs when we have a variable in our workspace that is *not defined* within a function. In that case, after failing to find the variable in the function's scope, R will look at the wider scope (the workspace) to see if it can find the variable before returning an error.

```
does_not_contain_z <- function(x = 1) {
  print(x = z)
}
```

This is somewhat dangerous, because in isolation, this function will fail. The *scope* of this function seems to know nothing about a `z` variable. Sure enough

```
does_not_contain_z()
```



**Error: object 'z' not found**

But if our workspace *does* contain a variable `z`, then R finds it when it expands its search to a wider *scope*

```
z <- "apple"
does_not_contain_z()
#> [1] "apple"
```

The difference between this example and the previous with all the `x` variables is that `print_x()` *did* contain a variable `x`, so R created a new `x` variable within that function's scope. Here, R widens its search through the scopes until it finds the variable `z` (or doesn't).

In this way, we can define some variables which will *always* be available to functions we create, with the disclaimer that they won't be used if our functions define their own versions of that name. These are commonly referred to as *global variables* because they are always defined. Anything defined in your *workspace* is essentially a *global variable* because this scope is wider than that of any function you create.

Using good variable names helps to avoid any conflicts — make your local variable names meaningful to their scope, and make sure you always define them with some value. R will always begin with the most *narrow* scope, so we could prevent the above scenario by defining a *local* `z` variable, even if it's meaningless

```
z <- "apple"
does_contain_z <- function(x = 1) {
  z <- "nothing"
  print(x = z)
}
does_contain_z()
#> [1] "nothing"
```



**It's important to note that R will only partially check that your function makes sense when you evaluate its definition. R will check that parentheses and curly braces are all matched up, and that the body of the function contains valid R code, but it won't check that variables or other functions are guaranteed to exist. This is due to the scoping we just discussed; variables and**

functions could very well be defined outside of the function you're defining in a wider scope, so there would be no way for R to check this when you define a new function.

This is actually a blessing when it comes to *recursion*; the concept of performing an operation which calls itself. We can define a function `factorial()` which calculates the product  $n \times (n-1) \times (n-2) \times \dots \times 1$ . In order to do this, we could either loop over the values of the input  $n$  from  $n$  down to 1, or we could take advantage of the fact that we need to essentially do the multiplication again with a smaller number many times. We need to stop when we get to 1, at which point we will return the final value. A function which does this looks like

```
## returns x*(x-1)*(x-2)*...*1
factorial <- function(x) {

  ## if the input is 1, just return the input
  if (x == 1) {
    return(x)
  } else {
    return(x * factorial(x - 1))
  }

}
```

Now, before we evaluate this function, R has no idea what the `factorial()` function looks like, but it *can* be defined, and that's good enough for R. As soon as we evaluate this function definition, there *is* a definition for `factorial()`, so the function can now be called. When we call it, R checks to make sure it knows what it's looking for (as per the above scoping rules) and finds it, since we've defined it. R has no issue calling it again, despite already in the process of doing so, since it creates another narrower scope for the new call to `factorial(x - 1)`. It continues doing this until the recursion exits, which we've allowed for by checking when the input is 1.<sup>37</sup> We can check how our recursive function stacks up against doing the calculation manually, and also against the built-in version from `base`.<sup>38</sup>

```
c(factorial(3) , 3*2*1, base::factorial(3) )
#> [1] 6 6 6
c(factorial(6) , 6*5*4*3*2*1, base::factorial(6) )
#> [1] 720 720 720
```

Next we'll look at how to expand the already wide variety of functions available to us, with *packages*.

## 4.2 Packages

### Package

A collection of functions and data which extend the functionality of the language.

A default installation of R comes with a wide variety of functions to complete an astonishingly diverse range of tasks, bundled together in the form of packages. The most basic of these is the `base` package which, along with the few pre-defined values

<sup>37</sup> It might help to know that the formal definition of a factorial function, often written as `x!`, defines the value `1! == 1`.

<sup>38</sup> `base::factorial()` is a good six times faster than our version, partly because it's written in C.

such as `pi`, defines the basic structures of the language.

We can most easily examine all of the `base` package functions by requesting a listing of them via the `help()` function

```
help(package = "base")
```

Alternatively, we could view the entire namespace using the Environment tab dropdown box (show the namespace of a specific package rather than the 'Global Environment'). This package contains many of the functions we've been using so far, including `print()`, `as.integer()`, and `typeof()`.

Common summary functions are also found in the `base` package, such as `sum()`, `mean()`, `length()`, and `nchar()`. Additionally, several packages are bundled with R installations which provide some essential functionality. The `utils` package contains many functions for interacting with the wider system, such as reading files. The `stats` package contains functions for performing statistical tests, amongst other things. The `graphics` package provides basic plotting functionality, while the `datasets` package includes a large number of built-in datasets which, since they are bundled with all installations, make for good testing/example values.

Frequently, the pre-installed functions are simply not enough for the analysis you wish to undertake. If you understand what you need to do and can write the function(s) yourself, then R is ready and waiting for you to do so. In many cases however, this effort has already been exerted by someone else and published as an available package, either on the Comprehensive R Archive Network (CRAN, which implements a variety of tests to ensure that packages will install on various operating systems without issue), Bioconductor (another open source repository of R packages specific to the study of genomic data) or a version control repository such as GitHub (for which testing is undertaken by the author, if at all, but has the advantage of a wider community input and faster development schedules).

To date, there are roughly 12,000 packages available on CRAN. There's a good chance that if you need a function to perform a well known task, there will be a package that already achieves that goal. The quality (in terms of freshness, accuracy, and ease of use) of said package is not necessarily guaranteed (though perhaps surprisingly few packages on CRAN are actually broken), but with open-source software there is always the opportunity to improve it. These packages are written by all levels of R users; from new users who fill a niche and create packages that may only be used a handful of times; to full-time R developers who push the boundaries of the existing language and create wonderful new ways to interact with data.

A list of the packages available on CRAN can be found here: <https://cran.r-project.org/web/packages/> and these can be sorted by name or publication date. I find the easiest way to find a relevant package however is to query a search engine with descriptive words related to what I'm trying to do, plus either `r` or `rstats`. This commonly brings up a discussion board or Stack Overflow question/answer where a package will be recommended.

### Dependency

An additionally required (separate) package which contains code/data to enable functionality within a package. A package may have many dependencies, and the R package framework allows these to be formally listed and installed when required.

When extending R by writing a package, one doesn't need to reinvent the wheel (or their own versions of **all** the functionality already available). Package authors can utilise functions from other packages, and when they do so that package becomes a *dependency* of their package. R keeps track of these as part of the package framework, so when a package is installed, all of the dependencies are also installed, so all of the functionality should be available.<sup>39</sup> For example, the `dplyr` package has the `Rcpp` package as a *dependency* since `Rcpp` contains functions for interacting with C++ code, which `dplyr` relies on. By allowing these dependencies, common functionality can be isolated into small packages which many other packages may then utilize.

To install a package from CRAN by name, `install.packages()` takes a package name as the first argument, and this is typically sufficient (provided you've set up CRAN or a local mirror of CRAN as the repository to search, otherwise R will likely ask you). To install the popular plotting package `ggplot2`, we need

```
install.packages("ggplot2") ①
```

- ① The first argument `pkgs` requires a string; the name of the package to install.

What will be shown in the `Console` (all going well) is a summary of the steps taken in installing a package; typically downloading, unpacking, installing dependencies, building, testing, and finally installing the actual package. All of this happens automatically, and rarely requires intervention. If this succeeds, you will be presented with something like

```
* DONE (ggplot2)
```

```
The downloaded source packages are in  
  '/tmp/RtmpTf0kV8/downloaded_packages'
```

Keep in mind though that installing a package only makes it available for *loading* (reading the function definitions into memory) and *attaching* (adding the function names onto the search list so that they can be looked up). We still need to *load* and *attach* the package before the functions can be used easily, which is best performed using the `library()` function

```
library(ggplot2) ①
```

- ① The first argument `package` can be a string (the name of the package to *load* and *attach*) but can also be a *bare* name (without quotes, as shown here) of the same.

<sup>39</sup> There are some exceptions to this, such as packages listed as `Suggests`; these aren't strictly required, but do offer further functionality, such as in examples, vignettes, or rarely used functions.

If you wish to *detach* a package (should you decide you really need to) this can most easily be achieved by restarting your session (RStudio shortcut: **Ctrl** + **Shift** + **F10**). There are functions to facilitate detaching a single package, but these require delicate handling to ensure that the *dependencies* are (or are not) also removed.



### A common package installation issue

When trying to install a package, it's common to have R complain that a package is not available for your version of R

```
install.packages("bloopbloop")
# Installing package into '/home/user/.BeyondSpreadsheets.Lib/BeyondSpreadsheets'
#> (as 'Lib' is unspecified)
# Warning message:
# package 'bloopbloop' is not available (for R version 3.4.3)
```

Sometimes this is simply because you've mis-entered the name of the package you want (e.g. ggplot instead of ggplot2). Other times it's because the package explicitly requires a newer version of R than you have. This is partly due to CRAN requiring new packages be built using the latest version of R, and that they enforce the same requirement on the users. In any case, it's a good idea to remain up to date with the latest stable version of R, which may resolve this issue.

Searching an archive of Stack Overflow R-tag questions (147,075 questions, spanning September 2008 to October 2016)<sup>40</sup> I see 270 matches for the phrase "is not available (for R version" which means this isn't a rare occurrence.

Once the package has been installed, the `install.packages()` step isn't required again until you wish to update the package to a newer version. The next time you wish to utilise some functionality from an installed package, the `library()` function (with the argument of the package name) is all that's required. With that said though, the `library()` call **will** be required each time you start a new session (for example, exit and re-open RStudio). This can be simplified by loading certain packages at the start of any new session, or by grouping together those `library()` calls at the start of your scripts.



When upgrading a package to a newer version, be very careful about reading the *changelog* which details any significant changes. New versions aren't always about fixing bugs, they can include improvements which involve changes to the way functions are called (different arguments, different defaults) and these can have undesired consequences to your code.

If we wish to work with 'in-development' packages not hosted on CRAN (such as GitHub) we are able to, but it's a good idea to understand the tradeoffs involved in doing so. Packages hosted on CRAN have undergone a detailed testing period which ensures they meet certain criteria (has some documentation, passes any consistency tests, installs correctly, refers to dependencies correctly, and many more). Packages on GitHub are in whatever state that particular developer has them in; there's no

<sup>40</sup> <https://www.kaggle.com/stackoverflow/rquestions>

guarantees of what the package is able to do or even does.

That said, it's a risk many people are happy to take, and as far as I am aware there aren't well known examples of things going horribly wrong. To install a package from GitHub, first we'll require the ability to build packages from their raw source (*binary* packages are often available from CRAN which are already compiled and ready to use). The state-of-the-art package for R developers is `devtools`, and you can read more about how to install its dependencies (which may include RTools for Windows or Xcode for Mac) here: <https://github.com/hadley/devtools>. Once you have that installed (using `install.packages("devtools")`) and loaded (`library(devtools)`) you can install packages from external sources, such as



```
# install.packages("devtools")
library(devtools)
install_github("username/repository")
```

- 1 Install the `devtools` package from CRAN. Quotes around the package name are necessary here.
- 2 Load and attach the `devtools` package.
- 3 Note that quotes around the package name aren't necessary here.
- 4 Note the name of this function does not involve a dot, but rather an underscore.

to install the package hosted at <https://github.com/username/repository>.

Whenever I show you some code which requires an extension package I will add the `install.packages("package")` and `library(package)` lines with the install step commented out; You only need to install packages once (until you upgrade them). If you haven't already installed the package you will need to uncomment that line (or type it into the Console yourself) and evaluate it.



### Package scopes for functions

There's no guarantees that two packages won't use the same name for a function. To specify exactly which package's scope we wish to take a function (or variable) from, we can prepend the name with the package and two colons. This applies when multiple packages provide functions with the same name, or we write a function and use the same name as a package's function. For example, if we wrote a `sum()` function, even though there is already one available in `base`, we can still use both as long as we are specific about requesting the one from `base` with the package prefixed

```
# A bad 'sum' function which just returns 0
sum <- function(...) {
  return(0)
}
sum(1, 2, 3)
#> [1] 0
# Explicitly use the 'base' version
base::sum(1, 2, 3)
#> [1] 6
```

This can also be useful if we don't wish to load an entire package just to use a single function, for example, we may wish to use `install_github()` from `devtools` just once, and we can do so with



```
devtools::install_github("username/repository")
```

### 4.2.1 How Does R (Not) Know About This Function?

One of the most commonly encountered errors from those new to R is finding that a function doesn't appear to be defined

```
someFunction()
```



**Error: could not find function "someFunction"**

Searching the Stack Overflow data archive (mentioned in [Packages](#)) for the above error message ("could not find function") produces 736 results — an entire half a percent of the questions asked included that error message in the question body. If you encounter this error, relax; you're not alone.

This occurs when R cannot find any definition for a function that appears in the expression to be evaluated. To understand how R *doesn't* know about some functions, we first need to understand how R *does* know about other functions.

When we discussed *scope* we saw that R will look through wider and wider scopes to find a value for a variable we ask for; starting within functions and expanding out to wider and wider scopes. Functions are searched for similarly. If we define a function in our *Workspace*, the code that it contains is stored in the *global namespace*. A *namespace* is merely the list of names of things (variables, functions, packages) within a given scope.

Just like variables, if you define a function *within* another function, then that defines the limit of its scope and the function can't be called on elsewhere. When you ask R to evaluate a function, it searches the *global namespace* for any functions you've defined there. If it can't find one, it expands the search to packages in the search path



#### Search Path

Packages which appear in the search path, and thus will be found when scope-searching, are those which you have *installed*, *loaded*, and *attached* using an installation function and the `library()` function. *Attaching* a package means that the contents of the package is finally added to the search path, and thus functions are available for you to use. Merely installing a package isn't enough for it to be available so easily.

This means that there is a hierarchy to the namespace. In a hypothetical situation where you *attach packageA* which contains a function `blop()`, then *attach packageB* containing a different version of `blop()`, then define your own version of `blop()`, R will *mask* the function each time (hopefully with a warning). Calling the function will take the definition from the narrowest scope; in this case your definition within the *global namespace*. If you *really* want to use a function from an installed package without *loading* the entire namespace (for example, to prevent

masking) you can specify this using a double colon (::) which will tell R where to search for a function, e.g. `packageA::blorp()`.

Another name for a namespace is an *environment*. We've seen that defining variables in the *global environment* causes them to appear in the Environment tab in the Workspace pane of RStudio. You may notice a dropdown box at the top of this pane which by default lists the Global Environment. Clicking this allows us to select a different *environment* to list, and lists the currently *loaded* and *attached* packages. Selecting one of these will show us the namespace for that package — typically a long list of functions.<sup>41</sup>

Back to the original question — what is wrong when R 'could not find function "someFunction"'? Hopefully now we see that R will have searched the global namespace and all loaded and attached package namespaces and not found a definition matching `someFunction()`. This means that either the package containing that function hasn't been installed, or merely hasn't been loaded.

The first thing to check is whether it has been loaded, and failing that, whether it has been installed at all. We can list all of the loaded namespaces with the `base` function

```
loadedNamespaces()
#> [1] "compiler"   "magrittr"    "graphics"    "tools"       "utils"
#> [6] "switchr"    "grDevices"   "stats"       "datasets"   "stringi"
#> [11] "knitr"      "methods"     "stringr"    "base"       "evaluate"
```

(or look through the list in the Environment pane's dropdown list).<sup>42</sup> If it isn't there, we can check if it is installed. The simplest way is to view the Packages tab in the same pane as the Help. If it's not listed there, then the package hasn't been installed. For now, there's plenty to see in the packages that come with R by default, but you won't break anything by installing another package and not using it.



### Masking of objects

We've now seen what happens when R can't find a function, but what about when it *can* and we try to use a package which also has a function with that name? For example, if we had written a `count()` function which returns the number of characters in a string (this is already a function; `nchar()` from the base package)

```
count <- function(string) {
  nchar(string)
}
count("abcde")
#> [1] 5
```

then we try to use the `dplyr` package, which also contains a `count()` function (with a much more useful implementation)

```
# install.packages("dplyr")
```

<sup>41</sup> These will commonly show up as `<Promise>` by virtue of the way that R *lazy loads* (delays formally defining) objects it doesn't necessarily need until they are used. If you start typing the name of one of these in the **Console**, RStudio's code-checking tools will likely be enough of a trigger to change the `<Promise>` to a fully-fledged function.

<sup>42</sup> Note: these may very well be different to what you get if you try this in your session, depending on what libraries you've used in your session so far.

```
library(dplyr)
#
#> Attaching package: 'dplyr'
# The following object is masked _by_ '.GlobalEnv':
#>
#>     count
# The following object is masked from 'package:switchr':
#>
#>     location
# The following objects are masked from 'package:stats':
#>
#>     filter, lag
# The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

we see that our `count()` function takes precedence because it is in the global environment (The following object is masked by `'.GlobalEnv'`: `count`). Some other functions are masked from other packages because `dplyr` is now higher on the search path (the more recently a package is loaded and attached, the higher its precedence).

## 4.3 Messages, Warnings, and Errors, Oh My!

It's inevitable that sooner or later something won't work and R will tell you about it. This will appear as text in the Console which does not begin with a prompt (>) and may be coloured differently to your normal input command text.

R has several different types of information it can send to you regarding the most recent evaluation:

### Messages

This doesn't mean anything went wrong, but the developer who wrote the function which raised the message allowed for R to tell you something. Perhaps some additional information to go along with your data result, or a notice that the function will behave differently in future versions. These will be printed to the Console immediately and won't stop your commands (or script) from running, but don't become complacent and ignore them either; messages about masking of functions can be very important.

### Warnings

Something, perhaps unexpected, has happened which is not critical, but is also not desired. This can be as innocent as NA values being introduced, or something as critical as your regression model not converging but exiting gracefully nonetheless. R will keep track of accumulated warnings and once your commands (or script) complete it will print them to the Console.

### Errors

These occur when something has failed entirely. These will usually cause your commands (or script) to exit prematurely and require careful inspection to

diagnose. These will be printed to the `Console` immediately and the `scope` reverted back to the global environment, making errors arising from within function calls somewhat difficult to diagnose (since any local variables will be destroyed by the time you regain control of the session).

### 4.3.1 Creating Messages, Warnings, and Errors



Follow along with these examples and observe the different outputs.

#### Messages

We can create a function that produces a message ourselves. The `message()` function takes any number of objects that it can convert to text and paste together. A single string is also quite useful

```
raiseMessage <- function() {  
  
  message("Keep going, you'll be a user in no time!")  
  
}
```

and we can call this function to observe the effect it has

```
raiseMessage()  
# Keep going, you'll be a user in no time!
```

Note that there isn't any context provided, the message is merely printed to the `Console`. This also doesn't start with a [1] since this isn't actually 'output'.

#### Warnings

Warnings behave differently. These are queued until the running expression is complete, at which point a summary is produced. If there are ten or fewer warnings produced, these are printed to the `Console`. If more than ten are produced then the number of warnings is noted with the instruction to use the `warnings()` function to view them all (the first 50 at least). If you accumulate at least 50 warnings (which can be surprisingly easy once we learn how to repeat operations) R will present a message stating

```
There were 50 or more warnings (use warnings() to see the first 50)
```

and, as it suggests, you can list these using the `warnings()` function.

We can create a function to produce a warning ourselves:

```
raiseWarning <- function() {  
  
  warning("Ouch! Please don't do that.")  
  
}
```

Evaluating this function, you may notice a different color in the output (depending on your theme settings)

```
raiseWarning()
# Warning message:
# In raiseWarning(): Ouch! Please don't do that.
```

Here we have some context that the text relates to a warning. It's a good idea to check the `Console` on completion of any evaluations.

### Errors

While you're unlikely to want to terminate your functions arbitrarily, we can create a function that does this. Unlike the other two commands, this one does not follow the same naming scheme (in which case this function would be called `error`). Instead, the `stop()` function performs a termination.<sup>43</sup>

```
raiseError <- function() {
  stop("I'm sorry, Dave, I'm afraid I can't do that")
}
```

which we can evaluate (albeit briefly) in the middle of several operations

```
a <- 2; raiseError(); b <- 3
```



**Error: I'm sorry, Dave, I'm afraid I can't do that**

Recall from [Working with R within RStudio](#) that commands separated by a semi-colon can be placed on the same line. Notice that the variable `b` is never assigned its value; check the `Environment` pane and observe that `a` is listed but `b` isn't; the execution was terminated as soon as the error was encountered. This can make life difficult when an error is encountered in the middle of a large block of processing as it becomes unclear whether certain pieces of code have been evaluated. When encountering an error, it's often a good idea to repair the issue then start fresh with a new session.

`stop()` can be used within functions or scripts when you really don't want the code to execute any further if a condition is met, in which case there is another function available which performs a test before exiting

```
smallValueCheck <- function(x) {
  stopifnot(x < 10) ①
  message("Look, the function made it this far!")
}
```

① < is the 'less-than' operator

If we evaluate this with a 'large' value (less than `10` is the condition required to avoid

---

<sup>43</sup> This is because the `stop` function does a lot more than merely terminate the code execution; R has ways to *catch* these errors and handle them in a more sophisticated way than merely terminating the execution, but this is beyond the scope of this book.

the error; execution will stop if the condition is *not* true)

```
smallValueCheck(20)
```



Error: x < 10 is not TRUE

yet if we try with a 'small' value, all is well

```
smallValueCheck(5)
```

```
# Look, the function made it this far!
```

Notice that no specific error message can be provided to the `stopifnot()` function itself. The help file for this (via `?stopifnot()`) shows that *all* of the arguments must be `TRUE` (not `NA`) for this to avoid terminating code execution. The statement that is output will indicate the *first* argument which was not `TRUE`; as many tests as you require can be separated by commas in the call to `stopifnot()`.

You'll inevitably have each of `Message`, `Example`, and `Error` show up at some time or other. Next, we'll learn how to deal with them.

### 4.3.2 How To Diagnose Them

Messages aren't of any concern, but they can be distracting if they show up too often or unsolicited. One common scenario in which they're found is when we load and attach a package using `library()` when the new package masks objects already defined. We saw this in [How Does R \(Not\) Know About This Function?](#) when loading and attaching the `dplyr` package, which looks like (if you're starting with a clean session)

```
library(dplyr)
#
#> Attaching package: 'dplyr'
#> The following object is masked from 'package:switchr':
#>
#>     location
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

Here we are provided with some potentially helpful information; that the namespace for the package `dplyr` has taken precedence in the search list over some other namespaces (recall: [Scope](#)). This can become annoying if you are already familiar with this and don't need to be reminded. While any message can potentially change in the future and you should really pay attention to these, they can be suppressed using the `suppressPackageStartupMessages()` function.

```
suppressPackageStartupMessages(library(dplyr))
```

General messages can be suppressed using the `suppressMessages()` function.

Since messages are merely printed to the `Console`, there isn't much else we need to do with these.

If we have produced more than 10 warnings we need to use the `warnings()` function to view a list of these. The warning text will be printed to the console, and will hopefully provide sufficient information to diagnose what went wrong.

Errors are usually a showstopper, so it is unsurprising that there's a more in-depth way to deal with these. If your code has stopped executing because of an error, you can request a *trace* of what R was trying to do (which functions it was calling) just before it reached that point. The  `traceback()` function prints the *call stack*; the expressions R was evaluating, possibly their line numbers relative to their source, and some of the values in use. The benefit of this is highly variable; sometimes this points to a malformed expression in an easily identifiable chunk of code. Other times it merely unpacks the function calls without really giving any indication of why it didn't work. The output of  `traceback()` is the sequence of calls (in reverse order, such that the call you actually made comes at the bottom) which led to the error. This will likely involve many functions you're unfamiliar with; the internal structure of the functions you use.

An example of this might be when we passed a large value to the `smallValueCheck()` function we defined earlier

```
smallValueCheck(100)
```



**Error: x < 10 is not TRUE**

We can examine where this error arose more specifically

```
 traceback()
#> 3: stop(msg, call. = FALSE, domain = NA) 3
#> 2: stopifnot(x < 10) at #3 2
#> 1: smallValueCheck(100) 1
```

- ③ The call which we evaluated.
- ② The internal step through which the error can be traced.
- ① The actual cause of the error; a `stop()` call within `stopifnot()`.

In each of these cases, we can use the `debugonce()` function to enlist the help of a *debugger*.<sup>44</sup> Providing the name of the function as an argument to `debugonce()` tells R to not merely execute the function body, but allow you to step through the lines at your own pace and try to learn exactly what is happening. If we wish to *debug* the `areaCircle()` function we defined earlier, we can pass this name to `debugonce()` which flags it for debugging the next time it is evaluated

```
debugonce(fun = "areaCircle")
```

<sup>44</sup> Not nearly as effective a tool as the name might suggest, but nonetheless a very useful tool; it won't actually remove the bugs, but it will help you isolate them.

If we now evaluate `areaCircle()` with some value, we are presented with a new prompt, this time starting with `Browse`

```
areaCircle(3)
debugging in: areaCircle(3)
debug at #1: {
  area <- pi * radius^2
  return(area)
}
Browse[2]>
```

This identifies that we are now in a *debug mode*. The highlighted line notes the next line that will be evaluated. We can press `Enter` or `N` (for next) to step through the lines of the body of the `areaCircle()` function. Try this yourself, and notice that the Environment pane now lists only the variables defined in this narrower scope (it also lists `areaCircle()` as the name of the environment).

To exit the debug mode, type `0` or press the Stop button, at which point you will be returned to the Global Environment and your normal Console.<sup>45</sup>

### ***On debugging code***

*“ If debugging is the process of removing software bugs, then programming must be the process of putting them in.”*

-- Edsger W. Dijkstra computer scientist and Turing Award recipient

Now we have our functions, and we know what to do if R complains about them, but do they work?

## **4.4 Testing**

If you or I were building a machine and finally finished tightening up the last screw, we wouldn't just assume it worked how we expect, would we? We might test that it behaves when we start it up the way we designed it to, feeding it some expected input and making sure we get the expected output. Making sure that no parts fall off or inputs get jammed.

In the same way, we wouldn't necessarily assume that our functions actually work before using them for an analysis, would we?

When a big company makes a machine, they'll test to make sure that it works as expected. A good company will also test to see what happens when the machine is used in other, perhaps less conventional ways. They'll often test that the safety mechanisms work when situations get dangerous, or push their machine to its breaking point so they know where that is.

The same principles apply for building functions; it's time to see how quickly they

<sup>45</sup> Pressing `ESC` may also exit debug mode.

break (or fail to).

Our `first_letter_formatted()` function worked well for the values I showed, but how flexible is it? Let's try some stranger inputs

```
first_letter_formatted("3.14159")
#> [1] "3.14159 starts with 3"
#> [1] "3"
```

Okay, that worked surprisingly well. How about a number, not a string?

```
first_letter_formatted(3.14159)
#> [1] "3.14159 starts with 3"
#> [1] "3"
```

Not bad. What if it's a variable that stores a number?

```
first_letter_formatted(pi)
#> [1] "3.14159265358979 starts with 3"
#> [1] "3"
```

Impressive. What about a number that doesn't start with 0 through 9 (a negative number?)

```
first_letter_formatted(-7)
#> [1] "-7 starts with -"
#> [1] "-"
```

The reason that all of these work is the our function calls `substring()` on the input argument `word`. Examining the internal code of that function shows that the first thing it does is convert its input into a character string using `as.character(text)`. This almost always works, so our function is fairly robust against odd inputs.

What about our `areaCircle()` function?

```
areaCircle("blue")
```



**Error: non-numeric argument to binary operator**

What about something that *looks like* a number?

```
areaCircle("5")
```



**Error: non-numeric argument to binary operator**

The reason is that the *exponentiate* function (`^`, raise to a power) can't handle non-numeric (e.g. character) arguments, unsurprisingly. If we needed it to be able to deal with this scenario without breaking, we would need to add some logic to ensure that `^` always receives a numeric value. We'll return to this in [Control Structures](#).

Testing is a powerful way to ensure that your functions work. Any time you create a function, it's a good idea to test it to make sure it's doing what you expect. We'll

return to how to formalise this in [Unit Testing](#).

## 4.5 Project 4.1 – Generalise a Function

Your boss saw your `first_letter_formatted()` function and thought it was great! Now they need it to be able to extract the first two letters, but only sometimes. You could

- write a new function which extracts the first two letters, and use whichever you need at the time; or
- add an argument to `first_letter_formatted()` which specifies how many letters to format.

Hopefully by now you recognise that the second option is superior. It reduces the amount of code which is duplicated which means fewer places for bugs to hide and fewer things to keep up to date.

Your task: re-write `first_letter_formatted()` so that a new argument `nLetters` can be provided (with a sensible default). The value of `nLetters` should be passed to the last argument of `substring()` so that it extracts the letters in positions 1 to `nLetters`. Don't forget to rename this new more-flexible function since it no longer extracts *just* the first letter all of the time. Document what your function does, what *types* it expects the arguments to take, and what the expected output of two test cases should be.

When you're done, ask a friend to read through your new function definition and guess what the resulting object will look like given some inputs.

## 4.6 Try It Yourself

 In the previous chapter we manually converted between Celsius and Fahrenheit temperatures, but we now know we can do this routinely with functions, so let's give it a go

```
# Convert Celsius to Fahrenheit
C_to_F <- function(temp_C) {
  temp_F <- (9L * temp_C / 5L) + 32L
  return(temp_F)
}

# Convert Fahrenheit to Celsius
F_to_C <- function(temp_F) {
  temp_C <- 5L * (temp_F - 32L) / 9L
  return(temp_C)
}
```

- ➊ Don't forget to add helpful comments.
- ➋ Naming arguments with meaningful names can be extremely helpful.
- ➌ Naming internal variables with meaningful names can save time trying to figure out what you meant to do.

- ④ Write calculations as clearly as possible. You'll thank yourself when you go to change them later.
- ⑤ Explicitly returning a value helps ensure you're returning what you think you are. Since `temp_C` is assigned on the previous line, if we left it with no implicit return statement, this function would print nothing when called.

We can now use these functions to convert *any* numeric temperature values

```
C_to_F(temp_C = 38)
#> [1] 100.4
```

The advantage of naming the variable along with its units is that incorrectly specifying one generates an error. In this next case, I have tried to use the argument `temp_C` in the function that is defined with an argument `temp_F`

```
F_to_C(temp_C = -40)
```



**Error: unused argument (temp\_C = -40)**

The correct argument leads to the function evaluating correctly

```
F_to_C(temp_F = -40)
#> [1] -40
```

The two scales produce the same value at -40°.

## 4.7 Summary

In this chapter we've built functions to simplify some calculations. These take inputs (*arguments*) and produce outputs. We've learned that R looks in different places to find variables, and that the same variable name can safely exist in multiple *scopes*. We've been introduced to packages as groups of data and functions that extend our capabilities, and seen the ways that R can tell us when something goes wrong.

You've learned that:

- Functions can be stored in variables just the same as any other data
- Abstracting expressions into functions means we can reuse them
- Functions don't alter data, the returned value needs to be assigned
- Function arguments are lazily evaluated
- Documenting a function outlines what **should** happen when it is used
- Functions can pass other functions in as arguments
- Arguments can be provided either by name or position
- The ... argument captures all non-formal arguments and passes them on
- Argument names will be partially matched if they are uniquely identifiable
- R searches wider and wider scopes when it can't find a requested variable
- Packages extend the language by providing more functions

New terms you've learned:

***return value***

the object that is produced by a function, typically provided by a `return()` function, and which can be assigned to another variable.

***abstraction***

representing some idea or set of operations in a more general notation, such as a function with a non-descriptive argument.

***scope***

the area of code for which a variable is relevant, such as a function, workspace, or package.

***namespace***

the list of names of variables/objects within a given scope.

***side effects***

any changes that persist outside of a scope, such as production of graphics, updating of data, or modification of the system, generally against the "functional" nature of R.

***global variable***

a variable which is defined in the `Workspace` and thus in the widest available scope, and so is available to any functions defined there also.

***local variable***

a variable which is defined in a narrower scope, such as in a function, which cannot be accessed from a wider scope.

***debug***

to step through the code defining a function at the level of its scope to determine the cause of errors.

Things to remember:

- Functions don't modify data; the return value of a function needs to be assigned to something.
- R will search through wider and wider scopes until it finds a requested variable (or doesn't).
- Every operation in R is a function call, even ones that don't look like functions (e.g. `+`).
- Default arguments can be a blessing and a curse. Know what they are (use the `help()` menu).
- Keep an eye out for messages, warnings, and errors. These don't always have the most informative outputs, but they can at least help you search for what has gone

wrong.

Now that we know how to create functions that take single values as inputs, it's time to learn how to create structures containing multiple values, which we can also send to functions as inputs. Because the creation of these structures is potentially a complicated operation, we'll need some way to take arbitrary values and work with them. You guessed it; more functions.



# I Want To Combine Data Values

**This chapter covers:**

- How to create structures containing grouped data; vectors, lists, data.frames, and more
- How these interact with missing data
- How to inspect and query these new structures.

So far we've only dealt with single-valued variables, but it's highly unlikely that your data looks so simple. We need a way to combine values into groups of data that together represent some larger concept. There are many ways to do this in R and the way we store collections of data can impact how we interact with it later, so let's see some of the different ways we can combine data.

## 5.1 Simple Collections

The simplest way to group together some values is with the built-in function `c()`. Feel free to refer to this however it best helps you to recall the name, but the words "concatenate", "combine", and "collect" are all good options.



Since `c()` is a function, it's a good idea to avoid naming your variables `c`, which may be tempting if you start with `a` and `b`. Granted, R is very clever at figuring out if you mean to use a variable or function, but having `c(c, c)` in your code just makes for a confused reader. Using "c" (the character) as a value is probably okay, though inadvisable, because we won't refer to it without the quotes.<sup>46</sup>

Together, multiple values form a *vector*, comprised of *elements*. We can create a vector of any basic ("atomic") type of value with one important rule; all elements **must** be of the same type: either numeric, integer, character, Date, etc... `c()` will combine all of its arguments into a single vector

---

<sup>46</sup> with the exception of functions from some packages, such as `ggplot2`, which use Non-Standard Evaluation (NSE) in their arguments, which doesn't require the quotes. That's a whole other story.

```
c(1.2, 2.8, 3.5)
#> [1] 1.2 2.8 3.5

c("cat", "dog", "fish")
#> [1] "cat" "dog" "fish"

c(TRUE, FALSE, TRUE)
#> [1] TRUE FALSE TRUE
```

Even a single value can be represented in a vector

```
c(6L)
#> [1] 6
```

This is the same output we receive when we evaluate `6L` on its own. In fact, it's impossible to create a literal 'single' value without it being in a vector.<sup>47</sup> Now we see why the R output always starts with a `[1]` — any non-zero number of elements forms a vector, even if just contains a single value.



The spreadsheet equivalent of a vector is a selection of one or more cells. Usually these are all in the same row or column and can be referred to with a *range*, e.g. A1:A8. Spreadsheet software will also typically warn you if values in a column of a table are of different types, though it will allow them (and pretend to work with them). An example of selecting a group of values in a spreadsheet is shown in Figure 5.1.

Figure 5.1. A selection of values in a spreadsheet.

	A	B	C
1	1.2	cat	TRUE
2	2.8	dog	FALSE
3	3.5	fish	TRUE

If we try to create an empty vector, R treats this as `NULL`

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

because it doesn't have a type to infer. We can explicitly create an empty vector of a specific type though using a type-creation function

```
numeric(length = 0)
#> numeric(0)

character(length = 0)
#> character(0)

logical(length = 0)
#> logical(0)
```

The arguments to `c()` need not be single values themselves; we can create a vector by combining shorter vectors just as easily

<sup>47</sup> R has no "scalar" types; even if you think you do, you don't.

```
c(1, c(8, 9), 4)
#> [1] 1 8 9 4
```

Some of the built-in datasets are simple vectors for convenience, such as `letters` (the lowercase English alphabet) and `LETTERS` (uppercase of the same).

If we try to print more elements than there is room for in one line of the Console, R breaks the line and continues on the next, showing the index of the vector at that point

```
print(LETTERS)
#> [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
#> [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

The `str()` function provides some useful information about an entire vector, listing the *type* of the elements (which must all be the same) and the indices of the elements

```
str(c(1.2, 2.8, 3.5))
#> num [1:3] 1.2 2.8 3.5

str(c("cat", "dog", "fish"))
#> chr [1:3] "cat" "dog" "fish"

str(c(TRUE, FALSE, TRUE))
#> logi [1:3] TRUE FALSE TRUE
```

So what happens if we *try* to use more than one type in a vector?

### 5.1.1 Coercion

If we try to create a vector with a combination of types, R performs automatic coercion and ensures that they **are** of the same type, by coercing to the most general type. Recall that the ordering from least general (most specific) to most general is

logical → integer → numeric → character

```
c(2L, 4.3, 0)    ①
#> [1] 2.0 4.3 0.0

c(TRUE, 5.3, 3L) ②
#> [1] 1.0 5.3 3.0

c(2L, 3L, "4")   ③
#> [1] "2" "3" "4"
```

- ① Coerced to numeric.
- ② Coerced to numeric.
- ③ Coerced to character.

The *type* of a vector created with `c()` is therefore the coerced *type* of its elements, which will be the most general of the types involved

```
typeof(c(2L, 3L, "4"))
#> [1] "character"
```

A vector created by combining smaller vectors, even if they have undergone coercion, will still be coerced to the most general type

```
c("1", 2L, 3L, c(4.0, 5.1))
#> [1] "1"   "2"   "3"   "4"   "5.1"
```

### 5.1.2 Missing Values

Missing values are allowed because they too have individual types; NA\_real\_, NA\_integer\_, NA\_character\_, NA (the default, is logical), so these can appear in vectors without issue

```
c(3L, NA, 5L)
#> [1] 3 NA 5
```

This is extremely useful for denoting when a value could have existed, but doesn't. If you specify a particular flavour of NA, then that will also be involved in the automatic coercion

```
c(2, 3, NA_character_, 5)
#> [1] "2" "3" NA "5"
```

The default NA is of type logical so it is usually coerced when used in a vector of non-logicals, but it is always printed as NA so it's not always clear that this has occurred.

The empty value NULL however cannot appear in a vector

```
c(3L, NULL, 5L)
#> [1] 3 5
```

R will simply skip over this element entirely.



Excel has no issue with leaving cells blank, and will even skip over them in certain calculations. In R, there is no exact equivalent for a blank element in a vector, but we can use the missing value placeholder NA. In a character vector you could include the empty character string "", i.e.

```
c("one", "two", "", "four")
#> [1] "one"  "two"  ""     "four"
```

but this isn't really *missing* as such, since "" is a valid value.

### 5.1.3 Attributes

A vector of data may consist of thousands of individual values, and we can store those in a variable which we name with some care. We might be able to encode *some* information about the values in that name (perhaps heights\_of\_giraffes) but often we need a way to attach another label to the data.

R has a mechanism for this that works well when used internally, but is somewhat fragile and I don't recommend you rely on it for anything you create yourself unless you fully understand the consequences. The mechanism is the attachment of *attributes* to objects (any object other than NULL) where additional information can be stored. For example, we might have a sequence of integers which represent something very important. We can store this sequence and refer to it with the variable x. The attributes() function lists any attributes of an object, and we see that

this new variable `x` has none

```
x <- c(1, 2, 3, 4, 5)
attributes(x)
#> NULL
```

We can attach an *attribute* with the label "important thing" to `x` and provide the string value "critical value" with the attribute assignment (there is a function `attr<-` to which R translates this operation)

```
attr(x, "important thing") <- "critical value"
attributes(x)
#> $`important thing`
#> [1] "critical value"

x
#> [1] 1 2 3 4 5
#> attr(,"important thing")
#> [1] "critical value"
```

Some functions will keep track of what these attributes are and pass them on to new objects

```
y <- x + 1
attributes(y)
#> $`important thing`
#> [1] "critical value"

y
#> [1] 2 3 4 5 6
#> attr(,"important thing")
#> [1] "critical value"
```

while other functions do not (hence the danger in using this feature)

```
z <- c(y, 12)
attributes(z)
#> NULL

z
#> [1] 2 3 4 5 6 12
```

There are valid uses for attributes, but most of them are handled by specific functions, so it is rare that you'll need to interact with them this way. Knowing they are there however can be vitally important, as we'll see.

## 5.1.4 Names

Storing data as a collection is very useful when all of the data represents a single unified concept, such as heights, counts, times, cities. Often though these values have another layer of definition which we would like to retain to identify which value goes where.

Thankfully, elements of vectors in R can have individual names associated with them, so we can recall which value represents which thing. We specify these as argument names to `c()`, which will create a vector from all of its arguments

```
c(apple = 5030, banana = 4011, pear = 4421)
#>   apple banana    pear
#>   5030    4011    4421
```

Notice now that the line has no [1] prefix, and that the names of the elements are listed above their values. R does an okay job at making sure that these stay aligned, even if their values are longer than their names

```
c(a = "antediluvian", b = "boisterously", c = "connoisseurs")
#>           a             b            c
#> "antediluvian" "boisterously" "connoisseurs"
```

If we look at the help file for `c()` (`?c`) we find that the definition of this function is `c(...)` which means it captures any named or unnamed arguments. This function therefore converts these argument names into element names for us.

There are limitations to what the names can be, and these are the same as the limitations on variable names. Names must start with a letter, not a number. They can start with a dot (.) as long as it's not immediately followed by a number. The rest of the name can consist of letters (upper and lower case) and numbers, but not punctuation (except . or \_) or other symbols. Here the dot is slightly more appropriate (as we'll see, even common in some cases). The following are all valid names for elements

```
c(high.value = 100, .score = 7, round2 = 26)
#> high.value      .score      round2
#>     100          7          26
```

whereas the following will fail

```
c(.2b = 6)
```



**Error: unexpected symbol in "c(.2b"**



#### Quote-defined names

Again we can bypass these restrictions by using quotes or backticks, but it means we need to use these when we come to accessing them.

```
c(".2b" = 6)
#> .2b
#>  6

c("one fish" = "red", "two fish" = "blue")
#> one fish two fish
#> "red"    "blue"
```

If we fail to name some elements (by not providing an argument name) then their name will be the empty string, which we won't see when we print the vector

```
c(apple = 5030, 4011, pear = 4421)
#> apple      pear
#> 5030    4011    4421
```

Calling `str()` on a vector with names produces a slightly different looking output to one without names

```
str(
  c(
    x = 7,
    y = 8,
    z = 9
  )
)
#> Named num [1:3] 7 8 9
#> - attr(*, "names")= chr [1:3] "x" "y" "z"
```

Now the output is prefixed with the word `Named` which tells us that this vector has named elements. The rest of the first line matches what we would receive from the unnamed vector creation

```
str(
  c(7, 8, 9)
)
#> num [1:3] 7 8 9
```

If we fail to assign a name, `str()` shows us that the empty string is being used

```
str(
  c(apple = 5030, 4011, pear = 4421)
)
#> Named num [1:3] 5030 4011 4421
#> - attr(*, "names")= chr [1:3] "apple" "" "pear"
```

The usefulness of this feature is limited to the values having some name you wish to retain. If we require more details to be stored along with our values, we will require a more complex structure, which we'll see shortly.

## 5.2 Sequences

It's likely you'll want to create a regularly increasing/decreasing vector of values at some point; a *sequence*. Perhaps as an index of distinguishable objects, or identifiers, the values one through five

```
c(1L, 2L, 3L, 4L, 5L)
#> [1] 1 2 3 4 5
```

can get tedious to enter manually, especially if you need a long sequence. There's an easier way; the `seq()` (sequence) function takes start and end value inputs and generates a regular sequence between the two

```
seq(from = 1, to = 5)
#> [1] 1 2 3 4 5
```

This has some clever default arguments too, namely that the sequence starts or ends at 1, so we can neglect one of these and still generate a sequence

```
seq(to = 5)
#> [1] 1 2 3 4 5
```

Leaving off all of the named arguments tends to work how you might expect (in many cases) but does so through a lot of guessing. If we try to generate a sequence from the input value 5 (since `from` is the first argument) to the default start value of 1, we get

```
seq(5)
#> [1] 1 2 3 4 5
```

which look suspiciously like `seq(from = 1, to = 5)`. In this case, the function guesses that we want an increasing sequence, and returns a sequence starting at 1 and ending at the input value when called in this way.



### The importance of explicit arguments

Explicitly naming the arguments `from`, `to`, and `by` can save you a lot of time trying to figure out why your sequence doesn't look as you expect. The defaults and internal code do a good job of guessing at what you're trying to create, but it's very easy to get caught up following a pattern. In our previous example, we created a sequence with `seq(5)` which assumes the default of `to = 1`. Since `from` is the first argument, and we didn't name one explicitly, this call assumes that we mean `from = 5`.

The fact that the sequence is increasing despite a request for a sequence from 5 to 1 is covered in the help page `?seq` with various examples of usage.

If we tried to modify our example and add a `by = 0.5` argument to generate the sequence in steps of 0.5, we would encounter an error

```
seq(5, by = 0.5)
```



Error: wrong sign in 'by' argument

This is our fault, not that of R; the code for `seq` covers various ways we might call the function, and does different things depending on which arguments are present. In this case, `from` is given the value 5, `to` is given the default value 1, and we have therefore attempted to generate a sequence from 5 to 1 in steps of +0.5.

Requesting that the steps go in the right direction works fine

```
seq(5, by = -0.5)
#> [1] 5.0 4.5 4.0 3.5 3.0 2.5 2.0 1.5 1.0
```

In particular, if you are using negative values, you may wish to be more explicit with your argument names

```
seq(-3)
#> [1] 1 0 -1 -2 -3
```

The *type* of sequence generated (`numeric` or `integer`) depends on the arguments, but with a bit of guesswork regarding what you might be hoping to achieve. When arguments are missing, an `integer` sequence is returned by default, even though the input is `numeric`

```
str(seq(5.0))
#> int [1:5] 1 2 3 4 5
```

If we change the step size using the argument `by` (even if it seems to be the same as the default) we can create a numeric sequence

```
str(seq(5, by = -1)) ①
#> num [1:5] 5 4 3 2 1
```

① Note the numeric 'by' value

Sequences don't need to be only whole numbers though. They don't need to start or end at integers either. We can create a regular sequence that increases in steps of 0.2 with

```
seq(from = 1.2, to = 2.0, by = 0.2)
#> [1] 1.2 1.4 1.6 1.8 2.0
```

and the `from` and `to` values don't need to be equally spaced either

```
seq(from = pi, to = 6)
#> [1] 3.141593 4.141593 5.141593
```

Again, the `seq()` function will make some guesses and assumptions about what it is you want. Notice that the largest value in this sequence isn't equal to the `to` argument, but another step larger would be too large.

When we have sequences that increase or decrease by 1, there is a shortcut function `:` which makes writing these sequences easier. This is one of the special functions (such as `+`) which can be written between two values

```
1 : 5
#> [1] 1 2 3 4 5
```

The spaces on either side of `:` are a matter of style, and most people will neglect them.



You may already be familiar with this notation from spreadsheet ranges, e.g. A1:A8, which selects cells 1 through 8 in the first column, as shown in Figure 5.2.

Figure 5.2. A sequence of selected cells in a spreadsheet.

A1:A8	
	A
1	21
2	22
3	23
4	24
5	25
6	26
7	27
8	28

You've also seen this in the output of `str()` on a vector

```
str(c(21, 22, 23, 24, 25))
#> num [1:5] 21 22 23 24 25
```

where the `1:5` indicates the range of indices of the elements.

We can create reverse sequences just as easily

```
5 : 1
#> [1] 5 4 3 2 1
```

and non-integer sequences (which step by 1) too

```
2.1 : 6.4
#> [1] 2.1 3.1 4.1 5.1 6.1

pi : 6
#> [1] 3.141593 4.141593 5.141593

3 : -3
#> [1] 3 2 1 0 -1 -2 -3
```

The use of `:` as a counter deserves a special note. Often we will want to create a counter that *iterates* over our values. We might have 3 names that we wish to perform a calculation using, and we could store these in a vector

```
patientNames <- c("Thomas", "Richard", "Henry")
```

We might access these using the indices 1 to 3. As we've seen, these indices form a regular sequence, so we may simply use `1:3`. If we're not entirely sure how many values we might need, we can generalise this so that the last value is equal to however long the vector is, using the `length()` function, so we could instead use

```
1:length(patientNames)
#> [1] 1 2 3
```

Now this is all well and good as long as we actually have some values in our vector, but if there's an update to the vector which removes them all, we can get into real trouble

```
patientNames <- c()      ①
length(patientNames)    ②
#> [1] 0

1:length(patientNames) ③
#> [1] 1 0
```

- ① No entries.
- ② The length of the empty vector is 0.
- ③ The sequence `1:0` produces the values 1 and 0.

We likely wanted our sequence to be empty, but since `:` accepts backwards sequences, this actually produces a sequence two values long. Trying to do something with the zeroth and first values of an empty vector will likely lead to some unexpected results.

In cases like this, it is almost certainly better to use a function which behaves as we expect. There are two additional functions, `seq_len()` and `seq_along()` which should be used instead of `:`  when counting things. Each of these takes a single argument — either the desired length of the sequence or something to count — and creates a regular sequence starting at 1

```
patientNames <- c("Thomas", "Richard", "Henry")
seq_len(length(patientNames))
#> [1] 1 2 3

seq_along(patientNames)
#> [1] 1 2 3
```

The big advantage of these is that if the thing you're counting the elements of becomes empty, the sequence generated from that is also empty

```
seq_len(0)
#> integer(0)

seq_along(c())
#> integer(0)
```



### Sequence Precedence

The sequence shortcut : has a place in the order of precedence, but it happens to sit between to other commonly used operators; one above, one below.

It's common to need to generate a sequence, but then not count the last element. Generating the sequence can be as simple as `1:3`. If the upper limit was stored as a variable, that might be

```
lim <- 3
1:lim
#> [1] 1 2 3
```

If we later needed to re-use the sequence, but not include the last value, we might try

```
1:lim - 1
#> [1] 0 1 2
```

but as you can see, the entire sequence has been shifted (now starts at 0 and is still 3 elements long). The order of these operations is actually to perform the sequence calculation *first*, then the subtraction (: is higher in the precedence list), so this is really equivalent to

```
(1:lim) - 1
#> [1] 0 1 2
```

If we want to change the to value of this sequence, we need parentheses around just that component

```
1:(lim - 1)
#> [1] 1 2
```

A different ordering happens however if we wish to raise the limit to a power

```
1:lim ^ 2
#> [1] 1 2 3 4 5 6 7 8 9
```

because exponentiation (^) is *higher* than : in the precedence list, so this is actually equivalent to

```
1:(lim ^ 2)
#> [1] 1 2 3 4 5 6 7 8 9
```

Again, strict use of parentheses will help avoid any issues over which operator takes precedence.

## 5.2.1 Vector Functions

Now that we have some vectors, we'll probably want to do something with them. We just saw that we can inspect our vectors and determine their length with the `length()` function

```
length(c(7, 8, 9))
#> [1] 3

length(c("dog", "cat"))
#> [1] 2

length(9:17)
#> [1] 9
```

We can calculate the sum of the elements just the same as if we passed them in one at a time

```
sum(c(10, 20, 30))
#> [1] 60

sum(10, 20, 30)
#> [1] 60
```

We can now calculate the mean (average) of several values (recall from our brief comparison between `sum()` and `mean()` in [Argument Name Matching](#))

```
mean(c(1, 2, 3, 4))
#> [1] 2.5
```

We can determine the minimum and maximum values in a vector

```
min(c(91, 23, 59, 44))
#> [1] 23

max(c(91, 23, 59, 44))
#> [1] 91
```

Even calculate some more advanced statistical quantities such as the standard deviation of a vector of values

```
sd(c("3.206171 5.369698 8.175691 2.739249 4.839496"))
# Warning message:
# In var(if (is.vector(x) || is.factor(x)) x else as.double(x), na.rm =
# > = na.rm): NAs introduced by coercion
#> [1] NA
```

Many functions will take multi-element vectors as input. Check the help file for any

function you intend to try this with though as it's not guaranteed.

## 5.2.2 Vector Math Operations

We need to be careful about mixing math operations with vectors, especially sequences since the lack of parentheses around the `:` operation means it's somewhat unclear what we mean.

Suppose we wanted to perform an operation that required a sequence 1 longer than `patientNames`. We might try

```
patientNames <- c("Thomas", "Richard", "Henry")
1:length(patientNames) + 1
#> [1] 2 3 4
```

but this no longer starts at 1. The reason, as we saw, is that the `:` operator has a higher precedence (recall: [Precedence](#)) than `+`, so that operation is performed first (generate the sequence). Recall that every group of values is a vector, even if it has a length of 1, so the `+` function definitely knows how to add two vectors when both lengths are 1. When one or both of the lengths isn't 1 however, some more tricks come into play.

The above example features a sequence of length three (`1:3`, three being the length of `patientNames`) and another of length one (the value 1). The `+` function needs to return a sequence the same length as the longest of these two, so it *recycles* (repeats) the shorter value enough times to create another sequence of length three. We therefore really have

```
c(1, 2, 3) + c(1, 1, 1)
#> [1] 2 3 4
```

When the lengths of the two vectors match, their elements can be added by matching up their positions



```
c(1, 2, 3)
# + + +
c(1, 1, 1)
# = = =
c(2, 3, 4)
```

 The spreadsheet way of thinking about this operation would be to imagine adding the first elements, then 'filling-down' the calculation to the other elements. Spreadsheets 'vectorize' the operation between the cells, as shown in Figure 5. 3.

Figure 5.3. Vectorized formulas in a spreadsheet performing addition between rows of cells.

	A	B	C
1	1	1	=A1+B1
2	2	1	=A2+B2
3	3	1	=A3+B3

When the lengths don't line up (one is not a multiple of the other), R complains

(rightfully so) as it isn't able to perform the operation cleanly

```
c(1, 2, 3) + c(2, 3)
# Warning message:
# In c(1, 2, 3) + c(2, 3): Longer object length is not a multiple of
#> shorter object length
#> [1] 3 5 5
```

Here the shorter vector (`c(1, 2)`) has undergone *some* recycling, but not a complete repetition, so the operation looks like



```
c(1, 2, 3)
# + + +
c(2, 3, 2)
# = = =
c(3, 5, 5)
```

If we are working with longer vectors, the shorter vector recycles as many times as necessary to reach the length of the larger vector

```
c(1, 2, 3, 4, 5, 6) + c(4, 3)
#> [1] 5 5 7 7 9 9
```

which is really



```
c(1, 2, 3, 4, 5, 6)
# + + + + +
c(4, 3, 4, 3, 4, 3)
# = = = = =
c(5, 5, 7, 7, 9, 9)
```

All of this works perfectly well if we have stored these vectors in variables too

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



**Don't underestimate this last line — we've just performed a *vectorised operation* whereby the elements of two vectors were added pairwise without us having to explicitly tell the computer to add the first pair, add the second pair, add the third pair (which in some other languages, you *do* need to do).**

Eventually this will become second nature, but for now try to mentally unpack these whenever you write them and you'll save yourself a lot of worry. The fact that we can perform vectorised operations along with automatic recycling means we are able to turn some simple expressions into less-than-simple results

```
x <- 1:10    ①
y <- c(1, 0) ②
x * y
#> [1] 1 0 3 0 5 0 7 0 9 0
```

- ① `length(x)` is 10.
- ② `length(y)` is 2, so it will be recycled to match the longer length.

The above is equivalent to multiplying the (recycled) vector `1, 0, 1, 0, ..., 1, 0` by the equally-sized sequence `1, 2, 3, ..., 10`, so every other value becomes zero.

Vector recycling becomes even more powerful when one of the objects involved in your operation is of length 1. For example, if we wanted to know which of the following values were greater than 5 we could test this without recycling as

```
c(6, 5, 4, 5, 7, 4) > c(5, 5, 5, 5, 5, 5)
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

or simply (by allowing the second vector to recycle)

```
c(6, 5, 4, 5, 7, 4) > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

If we have our vector stored as a variable, this becomes simple and compact

```
test_nums <- c(6, 5, 4, 5, 7, 4)
test_nums > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

We will make frequent use of this feature when we get to the loop construct. Too often people build a loop to perform some operation element-by-element when automatic vectorization will do this for you.

## 5.3 Matrices

Vectors are very handy for creating a single collection of grouped items, but sometimes we need more dimensions (e.g. locating points on a grid; we need both horizontal and vertical identifiers). If we wish to store the `x` and `y` coordinates of a set of points, we could store these each in individual vectors

```
x <- c(0.26, 0.57, 0.84, 0.72)
y <- c(0.01, 0.58, 0.82, 0.96)
```

but this is prone to losing track of which values correspond to which. Instead, we can create a *matrix* of values; a two dimensional grid of rows and columns. The function `matrix()` takes a data vector as its first argument, along with a specification of the number of rows and columns in which to store this data. We can store these `x` and `y` values by first combining them into a larger vector with `c()`, then specifying that we wish to store these as a matrix with 4 rows (one for each coordinate) and 2 columns (one for each of `x` and `y`)

```
xy <- matrix(data = c(x, y), nrow = 4, ncol = 2)
xy
#>      [,1] [,2]
#> [1,] 0.26 0.01
#> [2,] 0.57 0.58
#> [3,] 0.84 0.82
#> [4,] 0.72 0.96
```

The `matrix()` function will populate this matrix one column at a time by default. If your data is arranged such that it needs to be entered one row at a time, there is an

optional argument `byrow`. Note that since the input is a vector (created using `c()` here) the elements must *all* be of the same type. This is a restriction on the `matrix` object.

R will do the required division to spread out your data to the number of rows or columns you ask for

```
matrix(1:8, nrow = 2)
#>      [,1] [,2] [,3] [,4]
#> [1,]    1     3     5     7
#> [2,]    2     4     6     8

matrix(1:8, nrow = 4)
#>      [,1] [,2]
#> [1,]    1     5
#> [2,]    2     6
#> [3,]    3     7
#> [4,]    4     8
```

just as long as the division works out

```
matrix(1:7, nrow = 4)
# Warning message:
# In matrix(1:7, nrow = 4): data length [7] is not a sub-multiple or
#> multiple of the number of rows [4]
#>      [,1] [,2]
#> [1,]    1     5
#> [2,]    2     6
#> [3,]    3     7
#> [4,]    4     1
```

In this case we've once again encountered *recycling* — this time of our entire data vector which has been expanded to fit the requested shape (a rectangle with 4 rows).



This structure should begin to look rather familiar if you're accustomed to working with raw data in a spreadsheet as a table, as shown in Figure 5.4.

Figure 5.4. A table of values in a spreadsheet, similar to a matrix.

	A	B	C	D
1	1	3	5	7
2	2	4	6	8
-				

The `str()` function applied to a matrix tells us about each of the dimensions (note that still only a single type is reported, since all elements must be of this type)

```
str(
  matrix(1:8, nrow = 2)
)
#> int [1:2, 1:4] 1 2 3 4 5 6 7 8
```

These dimensions are also recorded in an *attribute* of the matrix with the label `dim`

```
attributes(matrix(1:8, nrow = 2))
```

```
#> $dim
#> [1] 2 4
```

where the number of rows and columns are stored in a vector (respectively).

### 5.3.1 Indexing

The `xy` matrix has the values of `x` as the first column, and the values of `y` as the second column. The rows and columns have descriptions printed alongside them; rows are indexed as `[row, ]` while columns are indexed as `[,col]`. R counts each of these, starting from 1, using numbers in both directions (horizontal and vertical). This is an important distinction if you are coming from a spreadsheet program which tends to label the rows with integers but the columns with letters.

In a spreadsheet, you may refer to a cell as A5 meaning the cell in the 5th row of the first column. In R, we use numbers in both positions, so that element would correspond to the `[5, 1]` (row 5, column 1) position in the matrix.

The `str()` function displays the *type* of the matrix (the common type of the elements, since they must all be the same) and the length of the rows and columns

```
str(xy)
#> num [1:4, 1:2] 0.26 0.57 0.84 0.72 0.01 0.58 0.82 0.96
```

Since the `data` argument is a vector, and all elements of a vector need to be of the same type, this coercion occurs before a matrix is even created

```
matrix(c("a", 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
#>      [,1] [,2]
#> [1,] "a"   "4"
#> [2,] "2"   "5"
#> [3,] "3"   "6"
```

Hence the single *type* printed by the `str()` function.

To make this easier to read, we can also specify the names of the columns using the `colnames()` function. Since we didn't tell the `matrix()` function what to use for column names when defining `xy` earlier, it doesn't have any

```
colnames(xy)
#> NULL
```

This function behaves differently however when we follow it with an assignment operator (`<-`) in that it can assign a vector to the names of the columns. Every operation in R is a function, and this is no exception; there is actually another function named `colnames<-` which performs this task, but R is clever enough to know that this is what you want to do

```
colnames(xy) <- c("x", "y")
xy
#>      x     y
#> [1,] 0.26 0.01
#> [2,] 0.57 0.58
#> [3,] 0.84 0.82
```

```
#> [4,] 0.72 0.96
```

If we try to assign names with a vector of the wrong length, R *won't* recycle the names

```
colnames(xy) <- c("z")
```



**Error: length of 'dimnames' [2] not equal to array extent**

Correctly setting the `colnames` of a matrix also sets another attribute; this one with the label `dimnames`. This has a more complex structure, but it essentially stores the names of the dimensions.

There is also a `rownames()` function (and corresponding `rownames<-`). There are limited cases where using this will be a good idea, but it's certainly possible. It behaves in the same way as `colnames()` and sets the names of the rows

```
rownames(xy) <- c("r1", "r2", "r3", "r4")
xy
#>      x     y
#> r1 0.26 0.01
#> r2 0.57 0.58
#> r3 0.84 0.82
#> r4 0.72 0.96
```

## 5.4 Lists

When we need to store things that aren't necessarily of the same length or type, we turn to `_list_s`. These are a more complex structure, but with that complexity comes flexibility and usefulness.

Technically, lists are vectors but it will quickly become clear that there's a lot more to them. We create lists with the unsurprisingly named `list()` function. We can create a very simple list with

```
list(7, 8, 9)
#> [[1]]
#> [1] 7
#>
#> [[2]]
#> [1] 8
#>
#> [[3]]
#> [1] 9
```

The output looks very different to that of a vector at first glance, but there are some similarities. We still have the `[1]` output, but now that appears next to *each* value, rather than just in front of the first. In addition, we now have a double-bracketed counter `[[1]]`, `[[2]]`, and `[[3]]`.

The argument to the `list()` function is ... which, as we saw in [Argument Name Matching](#), takes arguments of any (or no) name and passes them on to some internals. In the same way as we saw with the `sum()` function in that section, each *object* we

provide as an argument is taken individually, so in this case the `list()` function has created a list with three elements; 7, 8, and 9. Each of these, as we saw at the start of this chapter, is its own vector (of length 1) hence the [1] at the start of each listing.

We saw in [Simple Collections](#) that in a vector, the elements need to all be of the same type, else they will be coerced to such. Lists don't have this restriction, so we can create mixed-type lists

```
list(2L, "a", 5.2)
#> [[1]]
#> [1] 2
#>
#> [[2]]
#> [1] "a"
#>
#> [[3]]
#> [1] 5.2
```

without risking coercion.

The true benefit of lists comes when we store longer vectors. These individually follow the rules for their type (for vectors, elements must be of the same type or coerced to such) but different list elements can be entirely different in both length and type

```
list(
  c(1.2, 4.8),
  c(3L, 5L, 7L, 9L),
  c("cat", "dog", "mouse")
)
#> [[1]]
#> [1] 1.2 4.8
#>
#> [[2]]
#> [1] 3 5 7 9
#>
#> [[3]]
#> [1] "cat"    "dog"    "mouse"
```

We can mix-in shorthand notation for sequences, or anything we like (even more lists!). We can also name the elements of the list, which can be handy for identifying what the elements represent

```
list(
  nameMatrix = matrix(c("a", 2, 3, 4, 5, 6), nrow = 3, ncol = 2),
  commonValue = NA,
  index = 1:8
)
#> $nameMatrix
#>      [,1] [,2]
#> [1,] "a"   "4"
#> [2,] "2"   "5"
#> [3,] "3"   "6"
#>
#> $commonValue
```

```
#> [1] NA
#>
#> $index
#> [1] 1 2 3 4 5 6 7 8
```

Now we note that the generic [[1]], [[2]], [[3]] labels have been replaced with the element name, prefixed with a \$. This "dollar-name" format will be a useful way to access these elements later.

We noted that the `NULL` element cannot appear in a vector, but it's a different story for a list

```
list(
  x = c(4, 5, 6),
  y = NULL,
  z = c(7, 8, 9)
)
#> $x
#> [1] 4 5 6
#>
#> $y
#> NULL
#>
#> $z
#> [1] 7 8 9
```

the reason being that for a vector to hold a `NULL` element, it would need to coerce "nothing" into something (or vice-versa) which can't be done. A list however has no issue with one or more of its elements being "nothing" (`NULL`).

The `str()` function applied to a list reports each of the vector types and provides us with any names that have been defined

```
str(
  list(
    a = c(1.2, 2.3, 3.4),
    b = 2:5,
    c = c("alpha", "omega")
  )
)
#> List of 3
#> $ a: num [1:3] 1.2 2.3 3.4
#> $ b: int [1:4] 2 3 4 5
#> $ c: chr [1:2] "alpha" "omega"
```



**There isn't a spreadsheet equivalent of R's lists since spreadsheets don't specifically contain groups of data at all, more typically just lots of individual cells. Since a worksheet is just a grid of cells, and those cells can individually do as you need, you can spread out your data of any and all types wherever you like. This lack of structure tends to enable people to place 'tables' of data all over sheets, overlapped by graphs, calculations, and notes.**

## 5.5 `data.frame`'s

Keeping track of the positions along each list-element's vector for multi-faceted data would soon become insurmountable. Just as was the case when we expanded from a vector to a matrix, we can increase the dimensions of lists and create something more tabular.

The `data.frame` is the most common and useful type of object in R, but to truly understand it we've needed to see all of the predecessors first.

If we have a list of vectors, each having it's own (consistent) type, but all of the same length (or recyclable to the same length) then we can store that information in a table layout (rows and columns) using `data.frame()`

```
data.frame(
  list(
    x = c(1, 2, 3, 4),          ①
    y = c("a", "b"),           ②
    z = c(2.1, 9.3, 7.6, 1.1)
  )
)
#>   x y   z
#> 1 1 a 2.1
#> 2 2 b 9.3
#> 3 3 a 7.6
#> 4 4 b 1.1
```

- ① Here we have created a list with 3 elements (vectors) named "x", "y", and "z".
- ② The y vector is only of length 2, so it will be recycled to match the length 4 vectors.

R is clever enough to not actually need that `list()` call; we can simply send the vectors as arguments to `data.frame()`

```
data.frame(
  x = c(1, 2, 3, 4),
  y = c("a", "b"),
  z = c(2.1, 9.3, 7.6, 1.1)
)
#>   x y   z
#> 1 1 a 2.1
#> 2 2 b 9.3
#> 3 3 a 7.6
#> 4 4 b 1.1
```



This structure should be quite familiar if you've worked with spreadsheet tables before; column names and data are all here, but now instead of referring to the rows and columns with the A1 labels, we have the column numbers/names and row numbers (potentially names here too). A spreadsheet view of this might look like Figure 5. 5.

Figure 5.5. A spreadsheet table of cells.

	A	B	C
1	x	y	z
2		1a	2.1
3		2b	9.3
4		3a	7.6
5		4b	1.1

Technically the names for the vectors aren't required either, but it's dangerous to not supply them; without any names for the vectors they will be automatically created based on the way the values were supplied, with some care taken about spaces and other disallowed characters in the names, and made to be unique. Using the backtick/quote syntax for bypassing the naming rules leads to less than pretty column names which aren't particularly useful down the track

```
data.frame(c(1, 2), "'oh no!'")
#> c.1..2. X...oh.no...
#> 1      1    'oh no!'
#> 2      2    'oh no!'
```



Once again the special variable names can be applied here, as long as they are suitably quoted. It's quite common for spreadsheets to have columns named with spaces or numbers or punctuation, which R won't allow. One option is to use the quoting approach

```
data.frame(
  x = 21,
  "y variable" = 22,
  ".2b" = 23
)
#> x y.variable X.2b
#> 1 21          22  23
```

Here we see that R has allowed us to create the `data.frame`, but it draws the line at keeping those names in the improper format. Spaces and other unallowed punctuation are replaced with dots, while any names starting with a number (or dot-number) are renamed to start with an X.

If you really want to turn off this conversion (though it may very well mean your `data.frame` is not compatible with some other functions) you can do so with the argument `check.names = FALSE`

```
data.frame(
  x = 21,
  "y variable" = 22,
  ".2b" = 23,
  check.names = FALSE
)
#> x y variable .2b
#> 1 21          22  23
```

Since the columns are defined via a `list`, which is a group of vectors, each column can have a different *type* (though each element in a column must be of a consistent *type* since it is itself a vector). This is one of the ways in which

a `data.frame` differs from a `matrix`, where *all* of the elements in all of the rows and columns must be of the same *type*.

The requirement that the vectors all be the same length means that there will guaranteeably be some number of rows in the `data.frame` (even if that number is zero). Should we supply an incompatible set of vectors (for which the shorter can't be recycled to the longer length) then R complains and gives up

```
data.frame(x = c(3, 4), y = c("q", "r", "s"))
```



**Error: arguments imply differing number of rows: 2, 3**

R is perfectly okay with creating a `data.frame` with no rows at all

```
data.frame()
#> data frame with 0 columns and 0 rows
```

If we tried to add some named (but empty) columns we might think to try

```
data.frame(x = c(), y = c(), z = c())
#> data frame with 0 columns and 0 rows
```

but of course the empty vector `c()` is the same as `NULL`, which can't be compared to anything or contribute to a vector, so can't be used as a `data.frame` element. Instead, we can create 0-element named vectors and use these in the call

```
data.frame(
  x = integer(),
  y = character(),
  z = numeric()
)
#> [1] x y z
#> <0 rows> (or 0-length row.names)
```

"Empty" `data.frames` like this aren't entirely useless, as we'll see later. Being able to store the name of the columns even before any data is available can make working with `data.frames` much easier.



**We provide vectors to `data.frame()` to be used as columns, and these columns will have the type of each vector with one very important distinction; if we use a character vector then `data.frame()` will automatically convert this to a factor for us. If we create a `data.frame` using a character vector and a numeric vector, we see that the character column has been converted to factor**

```
char_and_num_df <- data.frame(
  x = c("apple", "banana", "carrot"),
  y = c(5, 6, 7)
)
str(char_and_num_df)
#> 'data.frame':      3 obs. of  2 variables:
#> $ x: Factor w/ 3 Levels "apple","banana",...: 1 2 3
#> $ y: num  5 6 7
```

**This causes a lot of confusion, but it's the default behaviour. To override this we**

`set stringsAsFactors=FALSE when creating the data.frame`

```
char_and_num_df <- data.frame(
  x = c("apple", "banana", "carrot"),
  y = c(5, 6, 7),
  stringsAsFactors = FALSE ①
)
str(char_and_num_df)
#> 'data.frame':      3 obs. of  2 variables:
#> $ x: chr  "apple" "banana" "carrot"
#> $ y: num  5 6 7
```

① Don't coerce character vectors to factor columns. The default for this argument is TRUE.

You can set this to be the default using the following option in your session before you create your `data.frame`

```
options(stringsAsFactors = FALSE)
```

Those column names are stored in an *attribute* (recall [Attributes](#)) but you may have noticed that R doesn't tell us this in the same way that it did when we printed a vector with an attribute. The reason is that R has some additional code it refers to when you ask it to print a `data.frame` to the Console. In order to see these, we need to use the `attributes()` function

```
attributes(mtcars)
#> $names
#> [1] "mpg"   "cyl"   "disp"  "hp"    "drat"  "wt"    "qsec" "vs"    "am"    "gear"
#> [11] "carb"
#>
#> $row.names
#> [1] "Mazda RX4"          "Mazda RX4 Wag"       "Datsun 710"
#> [4] "Hornet 4 Drive"      "Hornet Sportabout" "Valiant"
#> [7] "Duster 360"          "Merc 240D"        "Merc 230"
#> [10] "Merc 280"           "Merc 280C"        "Merc 450SE"
#> [13] "Merc 450SL"          "Merc 450SLC"      "Cadillac Fleetwood"
#> [16] "Lincoln Continental" "Chrysler Imperial" "Fiat 128"
#> [19] "Honda Civic"         "Toyota Corolla"   "Toyota Corona"
#> [22] "Dodge Challenger"   "AMC Javelin"     "Camaro Z28"
#> [25] "Pontiac Firebird"   "Fiat X1-9"       "Porsche 914-2"
#> [28] "Lotus Europa"        "Ford Pantera L"  "Ferrari Dino"
#> [31] "Maserati Bora"       "Volvo 142E"      ""
#>
#> $class
#> [1] "data.frame"
```

where we see the attributes `names`, `row.names`, and `class`. The first of these is the column names, while the next is the names of the rows. Row names are a peculiar feature and I don't recommend getting too attached to them. If we create a `data.frame` and print it (which R does by default)

```
data.frame(
  col1 = c("x", "y", "z"),
  col2 = c("q", "r", "s")
)
```

```
#>   col1 col2
#> 1     x     q
#> 2     y     r
#> 3     z     s
```

we see the rows labelled by their indices 1, 2, and 3. If we print the `mtcars` data set however, we see it too is arranged as a series of columns with names, but the first column appears to have no name and the indices are missing. This is because this particular data set has *labelled rows* (`row.names`). We can see these by asking for them

```
rownames(mtcars) ①
#> [1] "Mazda RX4"           "Mazda RX4 Wag"      "Datsun 710"
#> [4] "Hornet 4 Drive"      "Hornet Sportabout" "Valiant"
#> [7] "Duster 360"          "Merc 240D"        "Merc 230"
#> [10] "Merc 280"           "Merc 280C"        "Merc 450SE"
#> [13] "Merc 450SL"          "Merc 450SLC"      "Cadillac Fleetwood"
#> [16] "Lincoln Continental" "Chrysler Imperial" "Fiat 128"
#> [19] "Honda Civic"         "Toyota Corolla"   "Toyota Corona"
#> [22] "Dodge Challenger"    "AMC Javelin"     "Camaro Z28"
#> [25] "Pontiac Firebird"    "Fiat X1-9"        "Porsche 914-2"
#> [28] "Lotus Europa"        "Ford Pantera L"  "Ferrari Dino"
#> [31] "Maserati Bora"       "Volvo 142E"
```

① `row.names()` (with the period) also works. These aren't precisely the same functions though. Don't ask why.

which returns a vector of the row names. This is an odd design however — the names of the rows are not really *part* of the data, but are merely attributes of the data. I don't recommend following this pattern (though it's good to recognise it), instead I find it much more suitable to have these values as a character column in the `data.frame` itself.

The last attribute of the `mtcars` data set we saw in the output of `attributes()` was `class`, and this is a much more important attribute than it may appear at first glance.

## 5.6 Classes

We've already seen that R behaves differently when we ask it to print a `vector`, a `matrix`, or a `data.frame`

```
print(c(4, 5, 6))
#> [1] 4 5 6

print(matrix(c(4, 5, 6)))
#> [,1]
#> [1,] 4
#> [2,] 5
#> [3,] 6

print(data.frame(a = c(4, 5, 6)))
#> a
#> 1 4
```

```
#> 2 5
#> 3 6
```

but how does R know to do this in each case? We called the same `print()` function each time, right?

Objects may have an attribute `class` which labels the structure that object represents. A `matrix` has the `class` "matrix", and a `data.frame` has the `class` (10 points for guesses?) "data.frame".

We print these objects to the Console by calling the `print()` function with the object as the argument (as above). When R sees these calls, it first checks if the `class` attribute has been set for the object. If it has (say, to "someClass"), R looks through the functions it knows about and checks if any of their names match the pattern `print.someClass`. These extensions of a function which depend on the `class` are known as *methods*.

### Method

A function which only applies to a particular `class`. For example, the `print()` method for class "matrix". Different versions of the same *generic* function (in this case, `print()`) can be defined for different classes, and R knows to use these if the `class` attribute of the object passed into the first argument is assigned.

If R finds a `print()` method matching the class of the object, it uses that to print the object. If it doesn't find one, it calls `print.default()` — the default print method.

You can call any of the methods explicitly if you like, for example

```
d <- data.frame(x = 1:3, y = c(3.6, 2.7, 0.4))
print.data.frame(d)
#>   x   y
#> 1 1 3.6
#> 2 2 2.7
#> 3 3 0.4

print.default(d)
#> $x
#> [1] 1 2 3
#>
#> $y
#> [1] 3.6 2.7 0.4
#>
#> attr(,"class")
#> [1] "data.frame"
```



**It may become clearer now why I suggest **not** using a period in the name of your functions — does the `print.data.frame` method call `print()` for an object with `class` "data.frame" or does it call the `print.data()` method for an object with `class` "frame"? It's the former, though we can certainly construct the other scenario and become confused.**

This is a very simple system, and if it seems like it's not much more than saying "this

should behave slightly differently if it has this label on it", it's because that's a fairly accurate representation. There's nothing really stopping you from changing how R sees the `class` of an object (for your own sanity, you don't want to have to do this regularly) and thus how it thinks it should print it

```
x <- c(4, 5, 6)
class(x) <- "data.frame"
print(x)
#> NULL
#> <0 rows> (or 0-length row.names)
```

- ➊ Not a `data.frame`.
- ➋ But let's tell R that it is one by setting an attribute.
- ➌ Since `x` really isn't a `data.frame`, the `print.data.frame` method can't handle this.

We aren't limited to the classes that R already knows about — if we wish to create a more complex structure that behaves differently to a `data.frame` we can add additional attributes on top of existing ones. R will try each of these in order until it finds a method it can use, skipping over any that it doesn't recognise

```
class(d) <- c("specialObject", "data.frame")
str(d)
#> Classes 'specialObject' and 'data.frame': 3 obs. of 2 variables:
#>   $ x: int 1 2 3
#>   $ y: num 3.6 2.7 0.4

print(d)
#>   x     y
#> 1 1 3.6
#> 2 2 2.7
#> 3 3 0.4
```

If we want this behaviour to be different, we can define the method ourselves

```
print.specialObject <- function(x, ...) {
  cat("## I'm more than just a data.frame!\n")
  print.data.frame(x)
}

print(d)
#> ## I'm more than just a data.frame!
#>   x     y
#> 1 1 3.6
#> 2 2 2.7
#> 3 3 0.4
```

The rules surrounding how this can be done are beyond the scope of this book, but knowing that this occurs will be vitally important.

## 5.7 `tibble`'s

`data.frames` are a fundamental unit in the majority of R package functions. That said, there's certainly room for improvement with regard to a few of that structure's

features. One very useful extension in this space is the `tibble` package,<sup>48</sup> which provides a new structure `tibble`; similar to `data.frame` but with some carefully crafted improvements.

We first need to install, load, and attach the package

```
# install.packages("tibble")
library(tibble)
```

To quote from the help page `?tibble`, the advantages that `tibbles` have over `data.frames` are that a `tibble`:

- Never coerces inputs (i.e. strings stay as strings!).
  - (compare to `stringsAsFactors = TRUE`)
- Never adds `row.names`.
  - (these shouldn't be used. Compare to the `mtcars` dataset)
- Never munges column names.
  - (compare to `data.frame("my column" = c(1, 2, 3))` which creates a column `my.column`)
- Only recycles length 1 inputs.
  - (compare to `data.frame(x = c(1, 2, 3, 4), y = c("a", "b"))` which recycles the `y` column)
- Evaluates its arguments lazily and in order.
  - (copies of objects are only made when they need to, which saves computation time)
- Adds `tbl_df` class to output.
  - (`tibble` are identified by the `class` `tbl_df`, which means distinct methods can be applied)

In addition to these construction improvements, the package also provides an improved `print` method (`print.tbl_df()`) which itself has several improvements over the `print()` method for `data.frames`:

- Rows and columns which don't fit into the `Console` window aren't shown, rather they are noted as such. Compare this to the `print.data.frame()` method which wraps the output around after a certain number of rows, leaving us scrolling back and forth between blocks.

<sup>48</sup> Originally this was a part of Hadley Wickham's `dplyr` package as `tbl_df`, for table data frame, but pronounced 'tibble', and was extracted into its own package by Kirill Müller

- The total number of rows and columns is printed before the data.
- The type of each column is printed below the column name.
- Numeric precision is highlighted using shading (where supported).
- Negative numbers are coloured red (where supported).
- Missing entries are highlighted with yellow (where supported).

The `tibbles` printed in this book won't show these features, but if you're using a fairly recent version of RStudio then the output of

```
tibble(
  chars = letters[1:5],
  nums = -2:2,
  missing = c("x", NA, "y", NA, "NA"),
  precise = c(1.00002, 1.0002, 100.002, 10.200, 0.002)
)
```

should look something like figure 5.6.

**Figure 5.6. Helpful printing of different data features in tibbles.**

```
# A tibble: 5 x 4
  chars   nums missing  precise
  <chr> <int> <chr>     <dbl>
1 a         -2 x          1.00
2 b         -1 NA         1.00
3 c          0 y          100
4 d          1 NA         10.2
5 e          2 NA        0.00200
```

These features alone make `tibbles` a vastly superior object class, but there are further advantages still in how selection behaves.

If all of this isn't enough to convince you of the advantage that `tibble` has over `data.frame`, there's one more mindblowing feature that makes this package even more useful. We saw earlier that a `data.frame` is essentially a list of vectors, all of the same length. Those vectors can contain any basic *type* of value, be it numeric, character, Date, etc... `tibble` extends this concept to the idea of *list-columns*; rather than a vector of values, a `tibble` column consists of a list of objects. These can be anything, since they're in a list rather than a vector, so we can create a `tibble` with all sorts of data contained in a column, provided the length of each top-level *list* is the same.

We can create a `tibble` with a list-column just as easily as creating a `data.frame`

```
tibble(
  x = 1:3,                               ①
  y = list(letters, TRUE, mtcars) ②
)
#> # A tibble: 3 x 2
#>       x     y
```

```
#>   <int>
#> 1    1 <chr [26]>
#> 2    2 <lgl [1]>
#> 3    3 <data.frame [32 x 11]>
```

- ➊ The first column contains a vector of numeric values.
- ➋ The second column contains a character vector, a logical vector, and a `data.frame`, together stored in a list.

We can convert lists of lists to `tibbles` with the `as.tibble()` function

```
my_tbl <- as.tibble( 1
  list(      2
    x = 1:3, 3
    y = list( 4
      letters, 5
      TRUE,     6
      mtcars    7
    )
  )
)
```

- ➊ The `as.tibble` converts its contents to a `tibble`.
- ➋ The contents is a `list` of `lists` (a `list` of columns, some of which may themselves be `lists`).
- ➌ The first column is merely a sequence from 1 to 3.
- ➍ The second column is a `list-column` containing three elements;
- ➎ a character vector of length 26;
- ➏ a logical vector of length 1;
- ➐ and the `mtcars` dataset (a `data.frame`).

The `print` method for `tbl_df` is automatically called when we print this

```
my_tbl
#> # A tibble: 3 x 2
#>       x     y
#>   <int>
#> 1    1 <chr [26]>
#> 2    2 <lgl [1]>
#> 3    3 <data.frame [32 x 11]>
```

where we see the additional information available to us:

- The number of rows and columns are displayed (3 rows by 2 columns).
- The *type* of each column is displayed (`int` and `list`).
- The *type* and *length* of each element of the `list-column` is displayed (e.g. `<chr [26]>`).

If we have an object with lots of information, such as this `data.frame` made up of 26 rows and 26 columns

```
m <- as.data.frame(
  matrix(
```

```

  1:676, c(26, 26),
  dimnames = list(1:26, LETTERS)
)
)

```

we can `print()` it (I'm only showing the first six rows and that's bad enough)

```

head(m)
#>   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P   Q   R   S
#> 1 1 27 53 79 105 131 157 183 209 235 261 287 313 339 365 391 417 443 469
#> 2 2 28 54 80 106 132 158 184 210 236 262 288 314 340 366 392 418 444 470
#> 3 3 29 55 81 107 133 159 185 211 237 263 289 315 341 367 393 419 445 471
#> 4 4 30 56 82 108 134 160 186 212 238 264 290 316 342 368 394 420 446 472
#> 5 5 31 57 83 109 135 161 187 213 239 265 291 317 343 369 395 421 447 473
#> 6 6 32 58 84 110 136 162 188 214 240 266 292 318 344 370 396 422 448 474
#>   T   U   V   W   X   Y   Z
#> 1 495 521 547 573 599 625 651
#> 2 496 522 548 574 600 626 652
#> 3 497 523 549 575 601 627 653
#> 4 498 524 550 576 602 628 654
#> 5 499 525 551 577 603 629 655
#> 6 500 526 552 578 604 630 656

```

If we convert to a `tibble` however, rather than filling up our `Console` when we print it we are only shown the information which fits, with some notes on what was left out

```

as.tibble(m)
#> # A tibble: 26 x 26
#>   A     B     C     D     E     F     G     H     I     J     K     L
#> * <int>
#> 1 1     27    53    79   105   131   157   183   209   235   261   287
#> 2 2     28    54    80   106   132   158   184   210   236   262   288
#> 3 3     29    55    81   107   133   159   185   211   237   263   289
#> 4 4     30    56    82   108   134   160   186   212   238   264   290
#> 5 5     31    57    83   109   135   161   187   213   239   265   291
#> 6 6     32    58    84   110   136   162   188   214   240   266   292
#> 7 7     33    59    85   111   137   163   189   215   241   267   293
#> 8 8     34    60    86   112   138   164   190   216   242   268   294
#> 9 9     35    61    87   113   139   165   191   217   243   269   295
#> 10 10   36    62    88   114   140   166   192   218   244   270   296
#> # ... with 16 more rows, and 14 more variables: M <int>, N , O ,
#> #   P <int>, Q , R , S , T , U , V ,
#> #   W <int>, X , Y , Z

```

where we see that there are 16 rows and 14 columns (and their names) not shown. Resizing the console changes what's printed (the next time it's printed) too.



In the list of advantages of `tibbles` it was mentioned that they "never munge column names." Recalling that in order to have `data.frame` not convert spaces and punctuation to dots we needed to specify `check.names = FALSE`. We have no such trouble when working with `tibbles`

```

tibble(
  x = 21,

```

```

`y variable` = 22,
` .2b` = 23
)
#> # A tibble: 1 x 3
#>       x `y variable` ` .2b`
#>   <dbl>
#> 1 21.0    22.0    23.0

```

You may not have a need to create a `tibble` from scratch (there are very neat ways to do so) but once we start working with `data.frames` more we'll end up using functions which return this structure, so it's essential that we know about them.

### 5.7.1 Structures as Function Arguments

When we built and called functions in the last chapter, we only passed single values as inputs. Now that we know how to make more advanced structures, how do these work with functions?

You'll be happy to know that there's nothing new to learn; functions can be made to receive *any* structure as input (whether they're able to do something with the input is an entirely different matter). In fact, since there's no such thing in R as "just a value", you've actually been passing vectors (of length 1) to functions the entire time!

Most base R functions work well with vectors of length greater than 1 by default.<sup>49</sup> The `sqrt()` function, which calculates the square root of its input

```

sqrt(9)
#> [1] 3

```

will gladly perform the same operation on a (longer) vector of inputs, returning a new vector of outputs

```

sqrt(c(9, 16, 25))
#> [1] 3 4 5

```

The structural requirements of function arguments are specific to that function, but *should* be explained in sufficient detail in the help file for the function. If you're writing your own function, then it's up to you which input structures you wish to support or require. Make a sensible choice based on what your function *needs*; perhaps you *need* an entire `data.frame`, a list, or just a vector. Try to have as input the minimal amount of data for your function to achieve its task.

If you want to be rigorous about the input structure, consider adding some logic to test the input type with `class()` or `typeof()`. We'll see how to do this when we cover conditional evaluation.

Lastly, the *output* of a function (the argument to `return()`) can just as easily be any structure you require. A function can only return one object, but that need not stop you from building a `list` of individual pieces you wish to return. For example, the returned object from `lm(1~1)` (a trivial linear regression, the details aren't important here) is a

<sup>49</sup> They *usually* deal with vectors, since a length 1 value is still a vector.

list containing 11 parts

```
str(lm(1~1))
#> List of 11
#> $ coefficients : Named num 1
#> ... attr(*, "names")= chr "(Intercept)"
#> $ residuals : Named num 0
#> ... attr(*, "names")= chr "1"
#> $ effects : Named num 1
#> ... attr(*, "names")= chr "(Intercept)"
#> $ rank : int 1
#> $ fitted.values: Named num 1
#> ... attr(*, "names")= chr "1"
#> $ assign : int 0
#> $ qr :List of 5
#> ...$ qr : num [1, 1] 1
#> ... .- attr(*, "dimnames")=List of 2
#> ... .$. : chr "1"
#> ... .$. : chr "(Intercept)"
#> ... .- attr(*, "assign")= int 0
#> ...$ qraux: num 1
#> ...$ pivot: int 1
#> ...$ tol : num 1e-07
#> ...$ rank : int 1
#> ...- attr(*, "class")= chr "qr"
<truncated>
```

## 5.8 Try It Yourself

 Create your own vector of values. Can you mix different types (e.g. integers and characters)? Create another vector which has the values 10, 9, ..., 1. Did you use the `seq()` function? Can you use the `:shortcut`?

Try subtracting two equal length vectors and do the element-wise subtraction yourself to see if R arrives at the answer you'd expect. What if you multiply two vectors? What if they aren't of the same length?

Create a list that represents three qualities about yourself (e.g. your name, address, and date of birth). Split some of the parts of these qualities out into separate elements within each list element. Are the types what you expect?

Create a `data.frame` with three rows and four columns, where each row represents a person and the columns represent that person's name (character), age (integer), whether or not they like red wine (logical), and their height in meters (numeric). This might look something like

```
data.frame(
  name = c("Alice", "Bob", "Charlie"),
  age = c(21L, 43L, 19L),
  likes_red_wine = c(TRUE, FALSE, TRUE),
  height_m = c(1.65, 1.50, 1.75)
)
#>   name age Likes_red_wine height_m
```

```
#> 1 Alice 21      TRUE 1.65
#> 2 Bob 43      FALSE 1.50
#> 3 Charlie 19    TRUE 1.75
```

Now create the same `data.frame`, but acknowledge that you don't actually know whether or not Bob likes red wine by using `NA`. Assign this `data.frame` to a variable called `people`. Use `str()`, `print()`, `class()`, and anything else you know about to investigate this new object. Convert `people` to a `tibble` and print it. Do you notice anything? Did you remember to load the `tibble` package with `library(tibble)`?

## 5.9 Summary

In this chapter we've combined data into various types of structures with different benefits. We've seen how R coerces data types in certain structures (and doesn't in others) and how we can arrange our data to maintain certain attributes such as names and connected data.

You've learned that:

- You've been working with vectors all along; R has no concept of a 'scalar' value
- Vectors and matrices only contain a single type of data
- `NA` can be included in a vector, but not `NULL`
- Names can be attached to elements of a vector
- Integer-spaced sequences can be generated using `:`
- Not every function expects a vector or expects individual values
- When two vectors are involved in a comparison the shorter may be recycled to the longer length
- Lists can combine different types of data
- A `data.frame` will have a consistent number of rows in each column
- Different classes have different methods, such as `print.table()`
- The `tibble` structure is an improvement over `data.frame`

New terms you've learned:

### **vector**

a simple structure where all elements have the same type.

### **sequence**

a vector of regularly spaced elements.

### **recycle**

when the lengths of two vectors differ, one is repeated (recycled) to the longer length.

### **matrix**

grid of elements, all of the same type, laid out in rows and columns.

***list***

similar to a vector of vectors, but where each list element can be of a different type and length.

***data.frame***

a list of vectors where each vector is the same length, but may be of different types, laid out in a grid of rows and columns.

***class***

label of a particular structure which allows R to identify which functions (methods) to apply.

***tibble***

an improved `data.frame` with additional functionality (list-columns).

Things to remember:

- All elements of vectors and matrices must be of a single *type*, coerced to be the most general of those available.
- A `vector` or `matrix` can't contain `NULL`, but it can contain the relevant flavour of `NA`.
- A `list` can contain `NULL` as a top-level element, but it still can't be part of a `vector` or `matrix`.
- The `seq()` function will assume you mean to start at `1` if you don't specify enough arguments.
- The shortcut function `:` will gladly count backwards, whether you want to or not, e.g. `1:0`.
- Column names of a `data.frame` can't contain spaces or punctuation.
- A `tibble` is printed to the `Console` in a much more useful way than a `data.frame`.

Being able to create these structures is critically important, but it's unlikely that you'll always want to work with 100% of the data all of the time. We need to learn how to slice and dice these structures into smaller pieces.



# *I Want To Select Certain Data Values*

## **This chapter covers:**

- Manipulating strings
- How to select groups of values from different structures
- How to conditionally select values
- How to replace certain values
- A better way to manipulate `data.frames`; `dplyr`

Now that we've learned how to group values together into larger structures, it's going to be useful to access smaller components of those from time to time, be it a column from a `data.frame`, a row from a matrix, or part of a string.

Extracting data from larger structures is easily both one of the most common and most dangerous (because of various R "quirks") operations that you'll perform with the language. Knowing what the dangers are and how to expect them rather than being surprised by them is a defensive strategy, but there's value in understanding what's going on. Of course, R being an extensible language means that it only takes a bit more effort to replace the 'bad' bits with something that makes more sense, and sure enough that's where we're leading to.

## **6.1 Text Processing**

Joining together pieces of text is only one of the ways in which we can interact with strings. Often we need to extract out parts of a string, or replace one part with something else. This comes about most frequently when data is not quite of the form we require, or has been formatted in an odd way. Being able to identify particular patterns in text also means we can replace those wherever they appear. Once you begin working with thousands of strings at once, you'll be thankful you don't have to do any of the work manually.

### 6.1.1 Text Matching

Because strings are stored as collections of characters, we can think of them in that way and perform comparisons (recall [Comparisons](#)) between groups of characters and particular letters or patterns. One of the simplest questions to ask of a string is "does it start with...?". We can check this with the base function `startsWith()`

```
startsWith(x = "banana", prefix = "b")
#> [1] TRUE
```

There is a complementary function of course for checking the end of a string

```
endsWith(x = "apple", suffix = "e")
#> [1] TRUE
```

Note that this won't try to do anything clever like ignore whitespace; if a string starts with a space, then it starts with a space, not a letter

```
startsWith(" banana", "b") ①
#> [1] FALSE
```

- ① We can drop the argument names once we're familiar with the ordering.

Upper and lower case are also considered distinct, so

```
startsWith("Apple", "a")
#> [1] FALSE
```

The prefix or suffix we are checking can be any length, it need not just be a single letter

```
startsWith("carpet", "car")
#> [1] TRUE
```

Searching the start and end of a string are two specific cases of what we may wish to do more generally; search anywhere in the string. In that case we require a more advanced tool. Readers already familiar with Unix/Linux command line tools may recognise the `grep()` function which finds text which matches a pattern. Given a pattern, a text string, and the argument `value = TRUE`, `grep()` will return any text strings (the entire string) which contain the pattern

```
grep(
  pattern = "nan",
  x       = "banana",
  value   = TRUE ①
)
#> [1] "banana"
```

- ① This returns the matching strings provided we use the argument `value = TRUE`, otherwise this returns the vector indices of strings which contain the pattern.

In a similar fashion to the `startsWith()` and `endsWith()` functions, we can produce a logical result which indicates whether or not the pattern can be found in a string using the `grepl()` (1 for logical) function

```
grep1("app", "apple")
#> [1] TRUE
```

It's worth noting that all of the above functions are vectorised, so we can compare and process entire vectors of strings at once

```
grep1(pattern = "meow", x = c("meowing", "meowed", "homeowner"))
#> [1] TRUE TRUE TRUE
```

In these cases, with the `value = TRUE` argument, `grep()` returns only the strings which contain the pattern

```
grep("car", c("cradle", "cartoon", "carpet"), value = TRUE)
#> [1] "cartoon" "carpet"
```

and without this argument (or `value = FALSE`; the default) it returns the index corresponding to strings which contain the pattern (with only a single string, this is either 1 for a match, or `integer(0)` for no matches)

```
grep("car", c("cradle", "scary", "carpet", "Carroll"))
#> [1] 2 3
```

precedence These patterns are overly simple, and you would rarely require them. These functions will become vastly more useful shortly when we expand how these patterns can be structured more generally.



### A better way

As you may have guessed, there are more sophisticated third-party ways of achieving these tasks. The `stringr` package (another of Hadley Wickham's) provides simple tools to replace most of the functions we cover in this section.

```
# install.packages("stringr")
library(stringr)
```

For matching strings to patterns the `str_detect()` function essentially replaces `grep1()` and `str_view()` provides a very neat visualisation (see Figure 6.1) of which strings have matches (and where) inside the RStudio Viewerpane

```
# install.packages("htmlwidgets") ①
str_view(c("cradle", "scary", "carpet", "Carroll"), "car")
```

- ① This particular function I'm demonstrating here also requires the `htmlwidgets` library and its dependencies in order to show a HTML rendering of the string matches. It's not a formal dependency of `stringr` so it doesn't automatically get installed. To use this, you will need to install `htmlwidgets` yourself.

**Figure 6.1. Example of `str_view()` in the Viewer pane.**

```
cradle
scary
carpet
Carroll
```

`str_view()` will show only the first match, while `str_view_all()` will show all matches in

the case where there are multiple matches within a string.

### 6.1.2 Substrings

The pattern matching functions above are useful for determining **which** strings contain a pattern, but we are still left with the strings as a whole. If we wish to isolate part of a string then there are plentiful options to do so.

The `substring()` and `substr()` functions at first glance appear to do exactly the same thing; isolate a substring. They're subtly different, and capable of much more. Each takes a vector of strings (which can be just one, of course), and two positions; the `start/first` and `stop/last` indices of the substring to extract. The most visible difference between these functions is that for `substring()`, the default of `last = 10000000L` means that we may only need to specify the `first` argument to extract text from that position to what will surely be the end of our strings.

For example, to extract the first letters of some strings, we could use

```
substr(x = c("You", "Only", "Live", "Once"), start = 1, stop = 1)
#> [1] "Y" "O" "L" "O"
```

Note that I've specified both the `start` and `stop` values as 1 since I don't want to extract past that value. If we wished to extract the first three letters of a word, we would use

```
substr("carpet", 1, 3)
#> [1] "car"
```

We could save a tiny amount of writing if we wanted to extract the `last` three letters, by using the `substring()` function (with its default `last` argument)

```
substring("carpet", 4)
#> [1] "pet"
```

These examples are only useful if you already know which indices you require for extracting your substring. Often that's not the case, so we turn to more advanced functions.



#### More stringr

Again the `stringr` package provides better ways to extract substrings. One option being the `str_extract()` function. Until we have better tools to work with the patterns however, all we can do is extract out the part we already know; the pattern itself.

### 6.1.3 Text Substitutions

Replacing a substring is a common task, and there are many ways in which this can be achieved. The simplest is the `sub()` function. This is structured similar to the `grep()` function except now we also have a `replacement` argument. We can replace a pattern with some other text (which need not be the same number of characters)

```
sub(pattern = "pet", replacement = "tography", x = "carpet")
#> [1] "cartography"
```

This can be particularly useful for inserting things that are missing or incorrectly formatted

```
sub(" l", " L", "the Tower of london is located on the north bank of the Thames.")
#> [1] "the Tower of London is Located on the north bank of the Thames."
```

- ➊ By including a space in the pattern, we only match the start of words.

Note that `sub()` will only replace the first match to the pattern, despite both `london` and `located` being matches. If we wish to replace **all** matches, we need the `gsub()` function

```
gsub("mrs", "Mrs", "mrs smith, mrs jones, and mrs martin.")
#> [1] "Mrs smith, Mrs jones, and Mrs martin."
```

This can be useful for altering unwanted formatting, provided we remember to *escape* control characters inside argument strings (or use different versions)

```
gsub("\'", "\'", "He said 'Hello'")
#> [1] "He said \'Hello\'"
```



#### Have you installed `stringr` yet? You should.

The `stringr` package provides `str_replace()` and `str_replace_all()` for the single and multiple replacement scenarios.

This is also how we can deal with the 'commas in numbers' issue we noted back in [Numbers](#); replace commas with nothing (an empty string, "")

```
str_replace_all("1,200,000", ",", "")
#> [1] "1200000"
```

### 6.1.4 Regular Expressions

The string patterns we used so far have been simple strings themselves, but it's vastly more likely that you'll need to replace some group of letters, numbers, or symbols with certain characteristics. To do this we'll need to be able to refer to these groups using text, and in that case we turn to *regular expressions* or *regex*

#### Regular Expression (regex)

Patterns specified by a system of (possibly escaped) characters in order to capture text matches.

This topic is somewhat advanced, but it will be useful enough for you to be able to recognise when you need regular expressions, and what some easy examples look like. When we wish to extract or replace any single digit in a string we could do that with individual replacements easily enough if we know which digit we're looking for

```
sub(pattern = "3", replacement = "z", x = "xy3")
#> [1] "xyz"
```

but what if there's a general case where the digit could be any of 0, 1, ..., 9? In that case we can use a *regular expression* which matches to **any** digit. There are actually several ways to specify this: any of; \d (an escaped d, recall [Text \(Strings\)](#), where the backslash itself will need to be escaped); the grouping of digits [0123456789] (square brackets in a regular expression means 'any of these'); a shorthand notation for that group [0-9]; or the named digit class [:digit:]). Thus, even if we don't know exactly which digit we need to replace necessarily, the following will still work fine

```
sub(pattern = "\\d", replacement = "z", x = "xy8") ①
#> [1] "xyz"
```

① the backslash itself needs to be escaped here.

A detailed breakdown of regular expression selectors is available in the help menu for [?regex](#), and a brief table of some common examples is shown in Table 6. 1.

**Table 6.1. Some common regex selectors.[\[a\]](#)**

To select...	Use...	Examples
A specific letter\number	the letter\number, e.g. x	xY8 .
Digits	\d, [0-9], [:digit:]	xY8 .
Uppercase	[A-Z], [:upper:]	xY8 .
Lowercase	[a-z], [:lower:]	xY* .
Space	\s, [:space:]	xY8 .

[\[a\]](#) For more, see [?regex](#).

These can be woven together with the parts you do know to build up a full *regular expression*, and that can be used in the functions we've just seen such as `grep()` and `sub()`. For example, if we want to select a "c" or "C" (starting from anywhere in the string) followed by "ar" we could use the regular expression "[Cc]ar"

```
grep("[Cc]ar", c("cradle", "scary", "carpet", "Carroll"), value = TRUE) ①
#> [1] "scary"   "carpet"   "Carroll"
```

① [Cc] selects either C or c.

There are also ways to select the start (^) and end (\$) of a string, boundaries between words and spaces (\b), or any character at all (.). On top of all that, there are quantifiers which can follow a pattern to select how many times that pattern may appear in a valid match. These are briefly summarised in Table 6. 2.

**Table 6.2. Some common regex quantifiers for selecting multiple pattern matches.[\[a\]](#)**

To select a pattern... times	Use...	Examples
0 or 1	?	\d? matches xyz and xy8
0 or more	*	\d* matches xyz and xy8 and xy89
1 or more	+	\d+ matches xy8 and xy89
n	{n}	\d{3} matches xy789*

[a] For more, see ?regex.

Again we can use all of this in previously used functions to build very specific expressions

```
grep("[Cc]ar{2}", c("cradle", "scary", "carpet", "Carroll"), value = TRUE) ①
#> [1] "Carroll"
```

① r{2} selects two rs. Note that the entire expression must match.

There are many more options available to specify a regular expression, and hopefully it's apparent how powerful they can be in the right hands. With that flexibility comes complexity, so it's understandable that these aren't used to their full potential often enough.

To ease the difficulty of specifying the correct regular expression for a match, there are several tools which can help. The first is the `stringr` function `str_view()` which we saw in [Text Matching](#) which provides a visualisation of the matches in the RStudio Viewer pane. When we have multiple matches we can use the `str_view_all()` function. This is very handy for testing out regular expressions. For example, matching two digits can be achieved with

```
str_view_all("xy23yx878xyx9yxy", "\d{2}") ## two digits in a row
```

resulting in figure 6.2.

**Figure 6.2. Matching two digits in a row using str\_view(), this is shown in the Viewer pane.**

The `rex` package helps to build the regular expressions themselves by providing some shortcuts using more R-like syntax

```
# install.packages("rex")
library(rex)
#
#> Attaching package: 'rex'
# The following object is masked from 'package:stringr':
#>
#>     regex
```

This makes building up a complex regular expression much cleaner and easier to read. If we wanted to select the start of a string followed by any three letters regardless of

case, one digit, and then any number of lowercase letters, we can build that with

```
matching_regex <- rex(
  start,
  n_times(letter, 3),
  n_times(digit, 1),
  any_lowers
)
matching_regex
#> ^(?:[:alpha:]]){3}(?:[:digit:]]){1}[:Lower:]^*
```

The `(?: )` wrappers around each component have additional functionality that we're not making use of here, but that's how the `rex` package builds the expression. We can find matches to some strings then with `str_view()`

```
possible_strings <- c("xyzabc9?AB", "JDC8xyzabc", "SC?0abxyz")
str_view(possible_strings, matching_regex)
```

resulting in figure 6.3.

**Figure 6.3. Using the `rex` package to build up a regular expression, which can be used within `str_view()`.**

```
xyzabc9?AB
JDC8xyzabc
SC?0abxyz
```

Once you become familiar with regular expressions you'll find they become extremely powerful when working with text. Until now though we've been selecting parts of strings from either one or several strings. To be able to work with data in general, we'll need to be able to do similar things with the data structures we've created.

## 6.2 Selecting Components from Structures

Different structures require different commands to extract out specific components, and for various reasons, commands behave differently depending on how you're using them. A lot of this stems from trying to make your life easier, but with any guesses over what you're trying to achieve, inevitably structural consistency is the cost.

### 6.2.1 Vectors

If we have a vector of values and wish to extract just some of them, we need to select them by their index; their position in the vector. For example, if we have a vector

```
vec <- c(21, 23, 25, 27, 29)
```

and we wish to extract just the second element, we use the square-bracket operator (yes, a function under the hood, most frequently referred to by just `[` even though `R` requires the corresponding `]` to be present too) with the indices we want

```
vec[2]
```

```
#> [1] 23
```



### R is one-indexed

You may have heard that in other programming languages (such as C or python) they start counting elements at 0.<sup>50</sup> Not in R, which has the same numbering style as Fortran (a language at least a decade older than C), where we start counting the same way as everyday conversation; the first element is number 1, the second is number 2, and so on.

This operator can itself take a vector of values, so we can select multiple values at once

```
vec[c(2, 5)]
#> [1] 23 29
```

A sequence is just a vector, so we can extract a run of values all at once

```
vec[2:4]
#> [1] 23 25 27
```

If we wish to extract everything *except* a particular index, we prepend that index with a minus sign

```
vec[-2]
#> [1] 21 25 27 29
```

or exclude a vector of indices

```
vec[-c(2, 3)]
#> [1] 21 27 29
```

If we supply no indices, we get the entire vector

```
vec[]
#> [1] 21 23 25 27 29
```

although this is equivalent to just requesting the vector `vec` itself.

We don't necessarily need to have the vector saved in a variable to use this; we can extract values from a vector we have only just defined

```
c(32, 34, 36, 38)[c(2, 3)]
#> [1] 34 36
```

There is a second form of this operator named `[[]` (the object to be subset is followed by `[`, the subsetting details, then `]`), but this combination is most frequently referred to by the function name `[;` R just knows that the corresponding closing `]` will need to be present for the expression to make sense) which is only able to select a single element from a vector. Attempting to select more than one produces an error

```
vec[[c(1, 2)]]
```



**Error: attempt to select more than one element in vectorIndex**

<sup>50</sup> so that elements can be referenced in memory using an offset; the first element has offset 0, the next has offset 1, and so on...

One advantage of this however is that it removes names from vectors. If we have a named vector

```
named_vec <- c(x = 21, y = 23, z = 25)
```

we can extract a single element (with name) using

```
named_vec[2]
#> y
#> 23
```

or without its name

```
named_vec[[2]]
#> [1] 23
```

There are few occasions for which you will need to extract a single element from a vector this way, but it's a different story with more complex objects.

Identifying elements to be extracted based on their position (index) is fragile and should generally be avoided; the ordering may change, elements added or removed, etc... so it's unreliable to depend on these for extraction (though sometimes that's all we have). A more robust way to extract named element(s) is by combining [ or [[ with their name(s). To extract a named element from a vector we pass the name of the element(s) into [ or [[ as a character vector

```
named_vec[c("x", "y")]
#> x y
#> 21 23
```

or

```
named_vec[["y"]]
#> [1] 23
```

Again, [[ drops the names and only returns single elements.



### Unexpected selections

A surprising feature of index-based extraction is that using non-integers actually works. If we subset the built-in English alphabet vector letters by non-integer values

```
letters[c(1.3, 1.5, 1.7, 2.1)]
#> [1] "a" "a" "a" "b"
```

we see we have extracted the first element three times and the next one once.

In these cases the non-integer indices are *truncated* to just the integer part of the number (recall from [Specifying the Data Type](#) that `as.integer()` rounds towards 0) then the extraction is performed. This perhaps isn't so surprising; the selections we showed above used non-integer (strictly speaking) index specifications, e.g. in `vec[c(2, 5)]` we haven't used 2L and 5L.

Recall from [Simple Collections](#) that in R, indexes start at 1. Trying to select the 0 element doesn't produce an error or a warning, but it certainly doesn't exist

```
letters[0]
#> character(0)
```

Instead we've produced an *empty* result of type *character* (since *letters* is a character vector). Trying to select elements with an index greater than the length of a vector results in a *NA* value

```
letters[99]
#> [1] NA
```

Attempting to extract the element with index *NA* leads (as you may expect) to *NA*, but (as you may not expect), to not just one of them

```
named_vec[NA]
#> <NA>
#> NA NA NA
```

The reason for this is that *NA* (without being more specific) is actually a logical type, and as we will see, we can select elements with logical operators. Any comparison involving *NA* results in *NA*, so every element matches this criteria. We get just a single *NA* returned if we use a more specific missing value, such as the character version of *NA*, e.g.

```
named_vec[NA_character_]
#> <NA>
#> NA
```

Take some time to practice these extraction techniques because they're fundamental to further processing.

## 6.2.2 Lists

If we want to extract an element from a list we use the same syntax as with vectors. If we have a list with several elements

```
lst <- list(
  odd   = c(1, 3, 5),
  even  = c(2, 4, 6),
  every = 1:6
)
```

we can extract one or more of the elements with the `[` operator

```
lst[c(1, 2)]
#> $odd
#> [1] 1 3 5
#>
#> $even
#> [1] 2 4 6
```

which retain their names. Note that this returns the first and second *vector*, but the result is still a *list*

```
typeof(lst[c(1, 2)])
#> [1] "List"
```

In order to extract elements of those, we first need to specify *which* vector we wish to extract from (selecting only one). Trying to do this with `[` still returns a list,

```
typeof(lst[c(1, 2)][2])
#> [1] "List"
```

Recall that the double operator [ [ returns only a single element and drops its name. It also returns a vector from a list

```
lst[[2]]
#> [1] 2 4 6
typeof(lst[[2]])
#> [1] "double"
```

We can extract elements from this easily, since this is a vector

```
lst[[2]][2]
#> [1] 4
```



### Nested Lists

If you find yourself in the unfortunate position of dealing with *nested lists* (lists inside lists) then you may need to specify multiple values to [ [ in order to specify a nested element. For example, the following nested list is valid

```
nested_lst <- list(
  names = list(
    first = "Jonathan",
    last = "Carroll"
  ),
  fav_lang = "R"
)
nested_lst
#> $names
#> $names$first
#> [1] "Jonathan"
#>
#> $names$last
#> [1] "Carroll"
#>
#>
#> $fav_lang
#> [1] "R"
```

where the first element of this list is itself a list. If we need to extract the second element from the inner list, we can do that with

```
nested_lst[[c(1, 2)]]
#> [1] "Carroll"
```

We can extract named elements by specifying their name(s)

```
lst[c("even", "odd")]
#> $even
#> [1] 2 4 6
#>
#> $odd
#> [1] 1 3 5
```

and

```
lst[["even"]]
```

```
#> [1] 2 4 6
```

This helps somewhat with nested lists, since keeping track of the position indices gets confusing

```
nested_lst[[c("names", "last")]]  
#> [1] "Carroll"
```

In addition to a name in [] though, we can use the name to extract a single element using a dollar symbol \$ between the name of the list and the name of the element

```
lst$odd  
#> [1] 1 3 5
```

Note that this doesn't work for vectors, even if their elements are named

```
named_vec$a
```



### Error: \$ operator is invalid for atomic vectors

We need to be very careful with this feature, because R performs "partial matching" when it uses this dollar-name syntax, as it does for arguments (refer to [Partial Matching](#)). This means that you don't necessarily have to provide the *entire* name of the element you wish to extract (though, you should!), just enough of it that it can be uniquely determined.

```
lst$o  
#> [1] 1 3 5
```

This becomes an issue if you are expecting this to definitely work; there *may not* be enough of the partial name to uniquely determine the element

```
lst$e  
#> NULL
```

Note that this is the same result you obtain if the column simply doesn't exist

```
lst$z  
#> NULL
```

If you wish to be notified when this partial matching occurs (with a warning) then the option can be turned on by evaluating the following at the start of your session

```
options(warnPartialMatchDollar = TRUE)
```

which produces

```
lst$o  
# Warning message:  
# In lst$o: partial match of 'o' to 'odd'  
#> [1] 1 3 5
```

The partial-matching of names does not occur for the more-robust name extraction method

```
lst[["o"]]
```

```
#> NULL
```

unless you explicitly ask for it with the `exact` argument

```
lst[["o", exact = FALSE]]
#> [1] 1 3 5
```

This option can either be `TRUE` (only allow exact matches); `FALSE` (silently allow partial matches); or `NA` (allow partial matches but provide a warning when used).

Best practice of course is to not rely on a fragile feature and to supply the full name when using either convention.

Since we can specify `[[` extraction with names, this makes accessing the vector elements appear a lot cleaner

```
lst$odd[2]
#> [1] 3
```

and even with nested lists

```
nested_lst$names$last
#> [1] "Carroll"
```



#### **object of type 'closure' is not subsettable**

One of the most frequent trip-ups people encounter is a seemingly obscure reference to something not being "subsettable". This comes about when we have a function, say this one which creates a list with three values

```
make_a_list <- function() {
  return(list(x = 11, y = 12, z = 13))
}
```

We can evaluate this and generate the list as an object

```
new_list <- make_a_list()
```

We may want to extract one of the elements from this, in which case we could extract from this new variable

```
new_list$y
#> [1] 12
```

or, save ourselves the trouble of naming that object and extract it straight from the expression that generates it

```
make_a_list()$y
#> [1] 12
```

This is valid because R will break this down into the relevant steps before actually doing any of the operations; the function will be evaluated, *then* the subsetting operator `$` will be applied.

If however we forgot those parentheses which identify that we are calling the function `make_a_list()`, R will treat this as trying to subset the *function itself* (which is stored as an object). This isn't allowed though, so it fails

```
make_a_list$
```



Error: object of type 'closure' is not subsettable

The error message makes sense if you know that a *closure* is a proper name for a function object (which encloses both a series of expressions, and the environment from which it was created).

### 6.2.3 Matrices

When we have a structure with more than one dimension, such as matrix, specifying the indices that we wish to extract requires two values; one for the row and one for the column. This is still handled by the [ operator, but now includes a comma (,) between the row and column indices. If we have a matrix consisting of the first twelve letters of the alphabet (constructed using [ subsetting on the built-in letters vector)

```
mat <- matrix(letters[1:12], nrow = 3)
mat
#>      [,1] [,2] [,3] [,4]
#> [1,] "a"  "d"  "g"  "j"
#> [2,] "b"  "e"  "h"  "k"
#> [3,] "c"  "f"  "i"  "l"
```

we can extract a single element by specifying the row and column indices with the structure `m[row, col]`, so the element in the third row and second column is extracted with

```
mat[3, 2]
#> [1] "f"
```

Multiple elements can be extracted by supplying a vector to either the row or column position (or both), for example extracting the first *and* third rows and second column

```
mat[c(1, 3), 2]
#> [1] "d" "f"
```

Note that in each of these cases, a vector is returned. This is again R being more helpful than asked to be. One of the arguments to [ is drop and it defaults to TRUE. The consequence of this is that whenever the extraction result can be converted to a single dimension (a vector) it will be. Instead of returning a single-column or single-row matrix, R returns a vector. This may be what you really want to work with, but it's an inconsistency which makes programming difficult.



#### On R quirks

It's worth noting that in these examples I have taken the approach we've been sticking with in that the argument names are left out and just assumed by position. [ has formal arguments i and j which in theory would select the row (i) and column (j) elements of a matrix. In practice however, these explicit arguments are ignored and the selection is performed by position. Trying to swap these arguments around and extract with `mat[j = 2, i = 3]` will actually produce the same as `mat[2, 3]`.

If we select enough dimensions such that the result can't be reduced to just a vector, a

matrix is returned

```
mat[c(1, 3), c(2, 4)]
#>      [,1] [,2]
#> [1,] "d"  "j"
#> [2,] "f"  "l"
```



Some confusion may arise because we have asked for the first and third rows as well as the second and fourth columns, but the resulting matrix has row and column labels with 1 and 2 only. These aren't "names" as such, merely indices of the new matrix. Since the new matrix has two rows and two columns the indices cover this range.

The result of extracting part of a matrix will always be rectangular, having values in all rows and columns;<sup>51</sup> there is no way to create a 'triangular' matrix without the rest of the matrix being NA. This means that the resulting object (either a matrix, or a vector if drop = TRUE) will be formed by the intersection of the requested rows and columns. You will not produce a matrix with blank elements.

We can however specify that we want *all* elements in a row or column, specified by leaving out that selection (a space or merely nothing before or after the comma). To subset to all rows, we simply leave out the row selection

```
mat[, c(2, 3)] ❶
#>      [,1] [,2]
#> [1,] "d"  "g"
#> [2,] "e"  "h"
#> [3,] "f"  "i"
```

❶ The space before the comma isn't mandatory, but it can make the selection clearer.

When a matrix has names, such as the following

```
colnames(mat) <- c("col1", "col2", "col3", "col4")
rownames(mat) <- c("row1", "row2", "row3")
mat
#>      col1 col2 col3 col4
#> row1 "a"  "d"  "g"  "j"
#> row2 "b"  "e"  "h"  "k"
#> row3 "c"  "f"  "i"  "l"
```

we can use these names to perform the extraction

```
mat[c("row1", "row3"), c("col2", "col4")]
#>      col2 col4
#> row1 "d"  "j"
#> row3 "f"  "l"
```

We can also use the double [ ] syntax as long as we are extracting a single element

```
mat[["row2", "col2"]]
#> [1] "e"
```

---

<sup>51</sup> Square' matrices or single values being specific cases of 'rectangular'.



In each of the named-object extractions we have either used the explicit square-bracket syntax which takes the name of the element(s) we wish to extract, or we have used the dollar-name (\$) shortcut with the bare name of the element(s).

A complication of this is when our names bypass the usual rules and include spaces, or start with numbers, etc... These names need to be surrounded by quotes or backticks, but it otherwise works similarly

```
c(x = 7, "y variable" = 8, z = 9)["y variable"]
#> y variable
#> 8
list(x = 7, ".2b" = 8, z = 9)$`.2b`
#> [1] 8
```

## 6.3 Replacing Values

Extracting out a component of an structure is great if you want to isolate that, but sometimes we need to *change* a part of a structure. This requires us to select that part that we wish to update (using the indexing methods we've just covered). Typically this involves those same steps, but along with an assignment operator (<-). Most of the extraction operators provide this additional functionality in the form of another function. For [, there is a replacement operator (formally: [<-) which R translates to when it sees a subsetted object being assigned a new value.

If we have our vector `vec` and we wish not to extract the second element, but to *modify* it, we can do so with

```
vec[2] <- 99
vec
#> [1] 21 99 25 27 29
```

Note that evaluating the assignment operator still produces no output in the `Console`, so remember to `print()` (or simply evaluate) your object if you wish to inspect the changes.<sup>52</sup>

The new addition may cause the *type* of the object to be coerced

```
vec[2] <- "99"
vec
#> [1] "21" "99" "25" "27" "29"
```

An element of our matrix can be updated similarly

```
mat[3, 2] <- 99 ①
mat
#>      col1 col2 col3 col4
#> row1 "a"  "d"  "g"  "j"
#> row2 "b"  "e"  "h"  "k"
#> row3 "c"  "99" "i"  "l"
```

- ① The new element is a numeric type, but it will be coerced to character since that is the most general type of the two.

<sup>52</sup> As a reminder, surrounding a non-printing expression with parentheses will force it to print, such as `(no_print <- 2)`.

The replacements can also be *new* elements which previously didn't exist, with some caution over how these are produced. If we have a vector with length 5

```
vec <- 1:5
vec
#> [1] 1 2 3 4 5
```

we can request the eighth value

```
vec[8]
#> [1] NA
```

even though it isn't defined (and so it's set to NA). If we wish to *replace*(create) the eighth element, we can, on condition that we also create the sixth and seventh as well. We don't get a choice in this matter, they are automatically created for us, and set to NA

```
vec[8] <- 99
vec
#> [1] 1 2 3 4 5 NA NA 99
```

For a matrix, say

```
mat <- matrix(1:8, nrow = 2)
mat
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    3    5    7
#> [2,]    2    4    6    8
```

this simply isn't allowed

```
mat[8, 2] <- 99
```



**Error: subscript out of bounds**

We can replace several values at once using more vectors, as long as we are careful about matching the lengths of the elements we are replacing, and the lengths of their replacements

```
vec <- 1:8
vec[2:3] <- c(98, 99)
vec
#> [1] 1 98 99 4 5 6 7 8
```

allowing still for the case where recycling works

```
vec[1:7] <- 0
vec
#> [1] 0 0 0 0 0 0 0 8
```

If we don't wish to permanently overwrite the vector there is a handy function in the `base` package; `replace()` which does what the name suggests (without assigning the result) — it takes a vector `x` and replaces the elements at positions `list` with `values` (recycling if necessary), so we can replace the odd elements of `1:6` with 9 using

```
replace(x = 1:6, list = c(1, 3, 5), values = 9)
#> [1] 9 2 9 4 9 6
```

Similarly for a matrix, we can replace an entire column at once

```
mat[, 2] <- c(98, 99)
mat
#>      [,1] [,2] [,3] [,4]
#> [1,]    1   98    5    7
#> [2,]    2   99    6    8
```

For lists, we can request the second element with [[

```
lst <- list(
  odd   = c(1, 3, 5),
  even  = c(2, 4, 6),
  every = 1:6
)
lst[[2]]
#> [1] 2 4 6
```

We can update that element in the same way as with vectors

```
lst[[2]] <- c(8, 10)
lst
#> $odd
#> [1] 1 3 5
#>
#> $even
#> [1] 8 10
#>
#> $every
#> [1] 1 2 3 4 5 6
```

or with the dollar-name syntax

```
lst$even <- c(10, 12)
lst
#> $odd
#> [1] 1 3 5
#>
#> $even
#> [1] 10 12
#>
#> $every
#> [1] 1 2 3 4 5 6
```

We can even update single elements within those levels

```
lst$even[2] <- 100
lst
#> $odd
#> [1] 1 3 5
#>
#> $even
#> [1] 10 100
```

```
#>
#> $every
#> [1] 1 2 3 4 5 6
```

We can *remove* elements from a list by either de-selecting them with a minus sign

```
lst <- lst[-2]
lst
#> $odd
#> [1] 1 3 5
#>
#> $every
#> [1] 1 2 3 4 5 6
```

or by setting their value to `NULL`

```
lst$every <- NULL
lst
#> $odd
#> [1] 1 3 5
```

Attempting to add additional elements behaves similarly to vectors

```
lst[[4]] <- c("x", "y", "z")
lst
#> $odd
#> [1] 1 3 5
#>
#> [[2]]
#> NULL
#>
#> [[3]]
#> NULL
#>
#> [[4]]
#> [1] "x" "y" "z"
```

except that there is no placeholder value created automatically; the intermediate elements are given the value `NULL`.

Note that the new unnamed element (and automatically created elements with no names) are designated by their position in the list, while the named elements have their names.

We can similarly add more elements without worrying about how long the list already is by using the dollar name syntax

```
lst$greek <- c("alpha", "beta", "gamma")
lst
#> $odd
#> [1] 1 3 5
#>
#> [[2]]
#> NULL
#>
```

```
#> [[3]]
#> NULL
#>
#> [[4]]
#> [1] "x" "y" "z"
#>
#> $greek
#> [1] "alpha" "beta" "gamma"
```



### On unchanging data

We've said from the start that R doesn't modify data, but we've just gone through many examples which appear to do exactly that. What's happening here is that R is making a copy of our object, modifying *that*, then saving it back to the original variable name we had. This can use up a lot of resources if our objects are large, so don't be surprised if you're trying to update the thirteenth element of a `data.frame` column and R is taking a while.

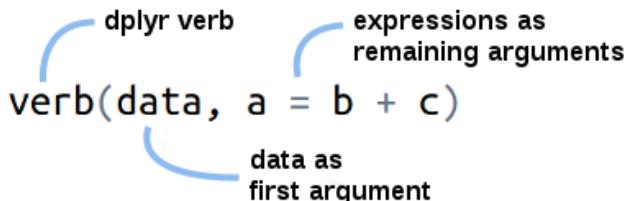
If you really need things done faster, the `data.table` package modified objects *in memory* rather than making copies of them.

## 6.4 `data.frames` and `dplyr`

When performing selections, extractions, and replacements on `data.frames`, the above mechanisms extend in more or less expected ways, with a whole slew of caveats and potential trip-ups. `$` or `[[` can be used to extract an entire column (since a `data.frame` is 'just' a list of vectors) or `[` can be used to extract elements in the same way as we would for a matrix. There is however, a better way.

The `dplyr` ("dee-ply-er") package by Hadley Wickham makes working with `data.frames` as simple as possible through a consistent and well-thought-out set of *verbs*

**Figure 6.4. dplyr verb construction**



`dplyr` is top on my list of recommended add-on packages for R. I start every session with it loaded, because I'm certain I'm going to use it. It provides a consistent interface of *verbs* for manipulating *tidy data* — data which is formatted in a proper, regulated manner, with one row per 'observation' and one column per 'quantity'. This is the structure of a `data.frame`, hence the `d` in `dplyr`. The functions in this package take one or more `data.frames` and produce modified copies of it/them with rows and/or columns selected, added, removed, or altered in some way.

In order to enable the use of this, we need to install, load, and attach the package. Installation of this package can take quite a while as much of it is actually written in more memory-efficient C code, which requires compiling.

```
# install.packages("dplyr")
library(dplyr)
```

Now that we have the functions available to us, it's time to learn some of the most important ones. These are the superstars of the `dplyr` package, and feature in the vast majority of analyses I've created. They're carefully constructed in the 'correct' style of writing a function; the short name describes what action the function performs with a *verb* (a "doing" word) and returns a predictable *type* of object.

Another feature of these verbs is that they consistently have `.data` as their first argument and ... as the remaining argument which captures expressions to be processed.

### 6.4.1 `dplyr` Verbs

The most important `dplyr` *verbs* you'll want to become acquainted with, which achieve the same goals as the above manipulations on 'smaller' structures, are:

- `mutate`: add or alter columns, defined by some calculation or operation on existing columns. Compares to selecting a subset of data and assigning new values to it; either a new column or an existing one.

```
head(
  mutate(mtcars, displ_l = disp / 61.0237)
)
#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb  displ_l
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1     4    4 2.621932
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1     4    4 2.621932
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1     4    1 1.769804
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0     3    1 4.227866
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0     3    2 5.899347
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0     3    1 3.687092
```

- `select`: keep only these columns. The order in which the selected variables are provided will be the new ordering of the columns. Compares to named subsetting of columns. Attempting to select a column that doesn't exist in the data will produce an error. Helper functions also exist to select columns that match criteria, such as `starts_with()`, `contains()`, and `everything()`. See `?select` for more.

```
head(
  select(mtcars, cyl, mpg)
)
#>          cyl  mpg
#> Mazda RX4      6 21.0
#> Mazda RX4 Wag   6 21.0
#> Datsun 710      4 22.8
#> Hornet 4 Drive   6 21.4
#> Hornet Sportabout 8 18.7
#> Valiant         6 18.1
```

- **filter**: keep only rows matching some logical criteria. These criteria can be passed in one at a time, or joined with a logical operator (and: &, or: |). Compares to named subsetting of rows.

```
filter(mtcars, cyl < 6, am == 1)
#>   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> 1 22.8   4 108.0 93 3.85 2.320 18.61  1  1     4    1
#> 2 32.4   4  78.7 66 4.08 2.200 19.47  1  1     4    1
#> 3 30.4   4  75.7 52 4.93 1.615 18.52  1  1     4    2
#> 4 33.9   4  71.1 65 4.22 1.835 19.90  1  1     4    1
#> 5 27.3   4  79.0 66 4.08 1.935 18.90  1  1     4    1
#> 6 26.0   4 120.3 91 4.43 2.140 16.70  0  1     5    2
#> 7 30.4   4  95.1 113 3.77 1.513 16.90  1  1     5    2
#> 8 21.4   4 121.0 109 4.11 2.780 18.60  1  1     4    2
```

```
## or equivalently filter(mtcars, cyl < 6 & am == 1)
```

- **summarise**: (also **summarize**) aggregate multiple values together and provide them to a function, returning a single result. This result will still be a **data.frame** so further processing may be required if you only want a value. Compares to the **stats** function **aggregate()**, particularly when the input is a grouped **tbl\_df**.

```
summarise(mtcars, mean(disp))
#>   mean(disp)
#> 1 230.7219
```

- **group\_by**: apply a grouping for which operations will be performed 'by group'. Compares to the internals of **aggregate()** which splits a **data.frame** by some grouping variable before computing the summary function.

```
head(by_cyl)
#> # A tibble: 6 x 11
#> # Groups: cyl [3]
#>   mpg cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#>   <dbl>
#> 1 21.0  6.00 160 110  3.90  2.62 16.5  0     1.00 4.00 4.00
#> 2 21.0  6.00 160 110  3.90  2.88 17.0  0     1.00 4.00 4.00
#> 3 22.8  4.00 108 93.0  3.85  2.32 18.6  1     1.00 4.00 1.00
#> 4 21.4  6.00 258 110  3.08  3.22 19.4  1     0     3.00 1.00
#> 5 18.7  8.00 360 175  3.15  3.44 17.0  0     0     3.00 2.00
#> 6 18.1  6.00 225 105  2.76  3.46 20.2  1     0     3.00 1.00
```

➊ Group the **mtcars** data by number of cylinders.

Notice that the output printed for the result of the **group\_by** operation looks different; **data.frame** doesn't support grouping like this, so this result is actually a **tibble** (recall **tibble** and thus uses **print.tbl\_df** for printing).

Groupings persist and are recognised by other verbs, so we can **summarise()** a grouped object to produce a grouped summary

```
summarise(by_cyl, disp_bar = mean(disp), hp_bar = mean(hp)) ⠁
#> # A tibble: 3 x 3
#>   cyl    disp_bar  hp_bar
#>   <dbl>      <dbl>   <dbl>
```

```
#> 1 4.00      105   82.6
#> 2 6.00      183   122
#> 3 8.00      353   209
```

- ➊ For each group, calculate the mean of the disp and hp columns.

Many more useful functions are provided by the `dplyr` package which I encourage you to discover. For now, these will provide ample opportunity to create rich analyses from your data in a clean and reliable manner.



#### On unchanging data

Note that these verbs don't alter the original input data (in these examples, the `mtcars` dataset) but rather return a new copy with the relevant operation performed. If you wish to preserve the returned value you'll need to assign it to a variable.

## 6.4.2 Non-Standard Evaluation

You may have noticed that in all of the above `dplyr` verbs we provided the arguments (frequently column names e.g. `mpg`) without explicitly stating that we meant "the data from the `mpg` column of the `mtcars` dataset". This may seem a little like magic, and that fact should make you slightly wary of it (but still impressed).

Don't let the name 'Non-Standard Evaluation' worry you, it just means that something special is going on behind the scenes to allow this to work. Internally, these functions check the local (defined within the function call) *scope* (recall [Scope](#)) to see if there's a column with that name, and if there is, it uses that data as the actual values in the calculation. This makes life a **lot** easier when we're working in the Console or evaluating line-by-line in a script because instead of typing things like

```
head(mutate(mtcars, hp_per_cyl = mtcars$hp / mtcars$cyl))
#>   mpg cyl disp hp drat    wt  qsec vs am gear carb hp_per_cyl
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1    4     4 18.33333
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1    4     4 18.33333
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1    4     1 23.25000
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0    3     1 18.33333
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0    3     2 21.87500
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0    3     1 17.50000
```

we can have some confidence that R will know to look in the `mtcars` data for these columns, so we can write

```
head(mutate(mtcars, hp_per_cyl = hp / cyl))
#>   mpg cyl disp hp drat    wt  qsec vs am gear carb hp_per_cyl
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1    4     4 18.33333
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1    4     4 18.33333
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1    4     1 23.25000
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0    3     1 18.33333
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0    3     2 21.87500
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0    3     1 17.50000
```

which makes it much easier to recognise what the calculation is doing. The danger

comes about if we want to change those columns programmatically, since we can't rely on this feature outside of the `dplyr` function, and so can't write something like

```
column_to_divide <- cyl
```



**Error: object 'cyl' not found**

(for the case where we might want to update that value later) since there is no variable `cyl` in our namespace. Instead, we need to store the column name as a character

```
column_to_divide <- "cyl"
```

We would like to be able to write the following, but `mutate()` thinks we're trying to divide a numeric column by a string, so this fails

```
head(mutate(mtcars, ratio = mpg / column_to_divide))
```



**Evaluation error: non-numeric argument to binary operator.**

Instead, we need to tell R to *unquote* this back into a *symbol* using the `rlang` package. The gritty details of this are fairly advanced and difficult to navigate, leading to somewhat more difficult to read code, but it does work

```
# install.packages("rlang")
library(rlang)
head(mutate(mtcars, ratio = mpg / !!sym(column_to_divide))) ❶
#>   mpg cyl disp hp drat    wt  qsec vs am gear carb      ratio
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1     4      4 3.500000
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1     4      4 3.500000
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1     4      1 5.700000
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0     3      1 3.566667
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0     3      2 2.337500
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0     3      1 3.016667
```

❶ `sym` converts our string into a symbol, while `!!` does the 'unquoting' to extract the stored value from it.

We'll use the normal semi-magical recognition of arguments feature, but it would be wise to remember that with this comes inherent limitations over where variables will be assumed to be found. If we actually have a variable `carb` then we need to take care about where R will be looking for it

```
carb <- 1000
head(mutate(mtcars, double_carb = carb * 2))
#>   mpg cyl disp hp drat    wt  qsec vs am gear carb double_carb
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1     4      4          8
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1     4      4          8
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1     4      1          2
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0     3      1          2
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0     3      2          4
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0     3      1          2
```

Name variables carefully.

If we do need to override this feature, we can be more explicit about where `dplyr` should look for certain variables (in the data we provide or in the environment?). If we wish to use the column `carb` in the above example (the default behaviour) explicitly we can refer to it as `.data$carb`

```
head(mutate(mtcars, double_carb = .data$carb * 2))
#>   mpg cyl disp hp drat wt qsec vs am gear carb double_carb
#> 1 21.0   6 160 110 3.90 2.620 16.46 0 1 4 4 8
#> 2 21.0   6 160 110 3.90 2.875 17.02 0 1 4 4 8
#> 3 22.8   4 108  93 3.85 2.320 18.61 1 1 4 1 2
#> 4 21.4   6 258 110 3.08 3.215 19.44 1 0 3 1 2
#> 5 18.7   8 360 175 3.15 3.440 17.02 0 0 3 2 4
#> 6 18.1   6 225 105 2.76 3.460 20.22 1 0 3 1 2
```

If instead we want to use the `carb` variable we also defined, we can refer to it as `.env$carb`

```
head(mutate(mtcars, double_thousand = .env$carb * 2))
#>   mpg cyl disp hp drat wt qsec vs am gear carb double_thousand
#> 1 21.0   6 160 110 3.90 2.620 16.46 0 1 4 4 2000
#> 2 21.0   6 160 110 3.90 2.875 17.02 0 1 4 4 2000
#> 3 22.8   4 108  93 3.85 2.320 18.61 1 1 4 1 2000
#> 4 21.4   6 258 110 3.08 3.215 19.44 1 0 3 1 2000
#> 5 18.7   8 360 175 3.15 3.440 17.02 0 0 3 2 2000
#> 6 18.1   6 225 105 2.76 3.460 20.22 1 0 3 1 2000
```



You may already be wondering how this deals with our special variables since we are using bare variable names which *may* in theory include spaces. If your column names have any irregular characters (spaces, punctuation, or start with a number or dot-number) then you can still surround these with backticks or quotes and `dplyr` will treat it like a bare name

```
d <- data.frame(
  x = 21,
  `y variable` = 22,
  `.2b` = 23,
  check.names = FALSE
)

select(d, `y variable`, `.2b`)
#> y variable .2b
#> 1 22 23
```

### 6.4.3 Pipes

The fact that the '`.data`' argument is *always* the first argument allows one further extension that, while it seems like a heavy departure from the way R has been written prior to now, makes reading and writing R code a great deal easier on the user.

The "pipe" operator `%>%` (a function, behind the scenes, of course) takes whatever is on its direct left and inputs that to the first argument of whatever is on its direct right (another function, either written as a function *name*; `foo` or a function *call*; `foo()`).

This looks like



```
x %>% y()
```

and can be read as "take x then apply the function y()" and can be thought of as equivalent to writing



```
y(x)
```

since x has now been provided as the first (in this case, only) argument to y(). If the function on the right takes more arguments, these are provided as if the first was already there.

This works for any type of data, so we can do 'simple' things too such as

```
c(100, 10, 1000) %>% ①
  log(base = 10) %>% ②
    mean()           ③
#> [1] 2
```

- ① Given a vector of values;
- ② Calculate the (vectorised) base-10 log() of these values, at this point equivalent to `log(c(100, 10, 1000), base = 10) = c(2, 1, 3).`;
- ③ Calculate the (vectorised) mean() of these values, at this point equivalent to `mean(log(c(100, 10, 1000), base = 10)) = mean(c(2, 1, 3)) = 2.`

This doesn't seem like a big deal until you have lots of functions all acting on top of each other, passing complicated objects into each other, such as

```
summarise(
  group_by(
    select(
      filter(mtcars, cyl > ④),
      mpg, hp, am
    ),
    am
  ),
  mileage = mean(mpg)
)
#> # A tibble: 2 x 2
#>       am mileage
#>   <dbl>
#> 1     0     16.1
#> 2 1.00    18.5
```

- ① filter mtcars to records with cyl greater than 4.
- ② select columns mpg, hp, am.
- ③ group\_by categories in the am column.
- ④ summarise by calculating the mileage as the mean of the mpg column.

which calculates the mileage for automatic (`am == 1`) and manual (`am == 0`) transmission vehicles in the `mtcars` dataset which have more than 4 cylinders (`cyl > 4`). This is difficult to unravel though; the `summarise` command is visually separated from the argument where the calculation is requested; the grouping by `am` sits out on its own away from the command itself. This is when the code is written with a generous application of linebreaks too — more often the calculation would be written on a single line as

```
summarise(group_by(select(filter(mtcars, cyl > 4), mpg, hp, am), am), mileage =
mean(mpg))
#> # A tibble: 2 x 2
#>       am mileage
#>   <dbl>
#> 1     0     16.1
#> 2     1     18.5
```

which is near indecipherable.

This "from the inside outwards" pattern of writing code is how functions are "supposed" to behave; the arguments are all in their correct places, but the *composition* (writing `f(g(x))`) makes it hard for us to read. R has no problem whatsoever, but why should we make it easiest for the computer when we're the ones who will get angry when we can't read it.

One solution is to split out the component function calls into more variables, i.e. one for

```
mtcars_filtered <- filter(mtcars, cyl > 4)
```

and so on, but this leads to the creation of a lot of variables that we only need for the next step, and naming things is hard.

The (vastly superior) alternative is the pipe, `%>%`. Instead of writing our function from the innermost data and adding functions around it, instead we can begin with the data then apply a function to it, then apply a function to whatever that produces, and so on, since the first argument to all of these functions is the data (what was just calculated). This becomes much easier to read, not least because the arguments to the functions stay within their parentheses. I hope you'll agree that the following is much clearer

```
mtcars %>%
  filter(cyl > 4) %>%
  select(mpg, hp, am) %>%
  group_by(am) %>%
  summarise(mileage = mean(mpg))
#> # A tibble: 2 x 2
#>       am mileage
#>   <dbl>
#> 1     0     16.1
#> 2     1     18.5
```

and produces the same output. We can see, line by line, what the steps in the processing are and the "flow" of the data through them. The `mtcars` data was inserted

into the first argument of `filter`, that function evaluated, that result was inserted into the first argument of `select`, and so on. This makes writing long "chains" of functions much simpler, and easier to read when we need to figure out what we were doing. It also helps to prevent mixing up of arguments and data since we can clearly see what is happening in each step.

It may help build your mental model of what's happening here if we use a temporary variable just this once. If we store `mtcars` as the variable `.` (just a period, which is indeed valid, though it won't appear in the `Environment` pane since it begins with a period)

```
. <- mtcars
```

Then use this explicitly as the first argument to `filter()`, storing that result back into `.`

```
. <- filter(., cyl > 4)
```

then explicitly using this as the first argument to `select()`, again storing that back into `.`

```
. <- select(., mpg, hp, am)
```

and so on, then we see that the `%>%` operator merely does this substitution for us. In fact, this is exactly what it does, and we can even use the `.` value in functions to refer to the left-side of `%>%` if we need to, such as filtering to every fifth row

```
mtcars %>
  filter(seq_len(nrow(.)) %% 5 == 0) ① ② ③
#>   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> 1 18.7   8 360.0 175 3.15 3.440 17.02  0  0     3    2
#> 2 19.2   6 167.6 123 3.92 3.440 18.30  1  0     4    4
#> 3 10.4   8 472.0 205 2.93 5.250 17.98  0  0     3    4
#> 4 33.9   4  71.1  65 4.22 1.835 19.90  1  1     4    1
#> 5 19.2   8 400.0 175 3.08 3.845 17.05  0  0     3    2
#> 6 19.7   6 145.0 175 3.62 2.770 15.50  0  1     5    6
```

- ① `nrow(.)` returns the number of rows of `.`, which in this case refers to `mtcars`.
- ② `seq_len(x)` creates the sequence `1, ..., x`.
- ③ `x %% y` calculates the remainder of dividing `x` by `y` as many times as possible, also known as 'modulo', for which the result will be `0` if `x` is an exact multiple of `y`.

There are many other advanced features of the pipe that we won't go into here, but support for this functionality is steadily growing amongst developers writing new packages.

The pipe is available automatically if you load the `dplyr` package, but its origin is in another package, `magrittr` named for the pun in Figure 6.2.

**Figure 6.5. René Magritte's 'La Trahison des Images' ('The Treachery of Images'). The French text is 'Ceci n'est pas une pipe' which translates to 'This is not a pipe'. <https://en.wikipedia.org/wiki/File:MagrittePipe.jpg>**



#### 6.4.4 Subsetting `data.frame` The Hard Way

`dplyr` should be your go-to for working with `data.frames` and `tibbles`, but it's also good to know how to work with the `base` operators relevant to these.

It's hopefully not too surprising that we perform extraction on `data.frames` in the same way that we do for a matrix. The same constructions (and caveats) apply here too. If we have our `data.frame`

```
my_df <- data.frame(
  x = c(1, 2, 3, 4),
  y = c("a", "b"),
  z = c(2.1, 9.3, 7.6, 1.1),
  stringsAsFactors = FALSE
)
```

then we can extract particular rows with `[a, ]`, where the blank entry after the comma indicates that we want all columns

```
my_df[3, ]
#>   x y   z
#> 3 3 a 7.6
```

Row names can also be used here the same as we did for a matrix, though I'll repeat that I don't recommend using row names for storing data. However, the `mtcars` data has this feature, with the rownames stored as strings

```
mtcars["Porsche 914-2", ]
#>          mpg cyl  disp hp drat    wt  qsec vs am gear carb
#> Porsche 914-2 26   4 120.3 91 4.43 2.14 16.7  0  1     5   2
```

We can extract particular columns with `[ , a]`, where the blank entry before the comma indicates that we want all rows

```
my_df[, 3]
#> [1] 2.1 9.3 7.6 1.1
```

We notice that the latter of these is a vector, despite extracting from a `data.frame`

```
class(my_df[, 3])
#> [1] "numeric"
```

The same `drop = TRUE` issue we found with `[` in [Matrices](#) is in effect here, but only when we select a single column; selecting a single row (every column of) still produces a `data.frame`

```
class(my_df[3, ])
#> [1] "data.frame"
```

If we wish to prevent this, we can explicitly request that the `drop` (to a vector) operation be avoided

```
my_df[, 3, drop = FALSE]
#>      z
#> 1 2.1
#> 2 9.3
#> 3 7.6
#> 4 1.1
```

This can make the expression look particularly odd if you extract one particular *row* and *do* want the result to be dropped to a simpler structure (in this case, a list which will preserve the fact that a single row extracted across columns of different types will contain different types)

```
my_df[3, , drop = TRUE]
#> $x
#> [1] 3
#>
#> $y
#> [1] "a"
#>
#> $z
#> [1] 7.6
```

Forgetting the additional comma results in a column being extracted (perhaps unexpectedly) and a warning

```
my_df[3, drop = TRUE]
# Warning message:
# In `[\$.data.frame`(`my_df`, 3, drop = TRUE): 'drop' argument will be
# ignored
#>      z
#> 1 2.1
#> 2 9.3
#> 3 7.6
#> 4 1.1
```

Extracting with `[`, `[[`, or `$`, partial matching of names, returning `NULL` for missing columns, and related issues all apply to `data.frame`'s.

The dollar-sign extraction and replacement methods work just the same too (and are more robust to re-orderings of data than using positional indices). We can extract columns

```
my_df$x
#> [1] 1 2 3 4
```

or replace elements within them

```
my_df$z[2] <- 99
my_df
#>   x y     z
#> 1 1 a  2.1
#> 2 2 b 99.0
#> 3 3 a  7.6
#> 4 4 b  1.1
```

and we can use names to achieve this in the same way

```
my_df[1, "x"] <- 99
my_df
#>   x y     z
#> 1 99 a  2.1
#> 2 2 b 99.0
#> 3 3 a  7.6
#> 4 4 b  1.1
```

We can also add new columns in this way

```
my_df$new_column <- c("red", "blue", "green", "white")
my_df
#>   x y     z new_column
#> 1 99 a  2.1      red
#> 2 2 b 99.0      blue
#> 3 3 a  7.6      green
#> 4 4 b  1.1      white
```

I personally do still use the dollar-name syntax for extracting a single column from a `data.frame` when working in the console or with data where I **know** a certain column will have a certain name, as it's easily the fastest way to do so. When programming inside a function however, it's much safer to use the square-bracketed column selection tools.

## 6.5 Replacing `NA`

One very frequently used pattern is replacing `NA` values with something. Perhaps these are more meaningful to your analysis as `0` values, or some character string. We can combine conditionally selecting with replacement to achieve this, but we need to be careful.

Simply comparing our data to `NA` won't work, since a comparison with `NA` always results in `NA`, and as we saw earlier, selecting the `NA` element just selects all of them. Instead, we can use the `is.na()` function we saw in [Comparisons](#)

```
x <- c(7, NA, 9, NA)
is.na(x)
#> [1] FALSE TRUE FALSE TRUE
```

Since this produces a logical vector, we can use it to subset some data, then replace those NA values with something else. If we have a `data.frame` with NA values

```
d_NA <- data.frame(x, y = 1:4)
d_NA
#>   x y
#> 1 7 1
#> 2 NA 2
#> 3 9 3
#> 4 NA 4
```

we can replace NA with 0 by selecting the rows for which x is NA, and the column x

```
d_NA[is.na(d_NA$x), "x"] <- 0
d_NA
#>   x y
#> 1 7 1
#> 2 0 2
#> 3 9 3
#> 4 0 4
```

`is.na()` also has a `data.frame` method (recall [Classes](#)) so we can actually pass the whole `d_NA` object in and this will return a logical `data.frame`

```
d_NA <- data.frame(x, y = 1:4)
is.na(d_NA)
#>   x     y
#> [1,] FALSE FALSE
#> [2,] TRUE FALSE
#> [3,] FALSE FALSE
#> [4,] TRUE FALSE
```

which means we can use this condition for replacement

```
d_NA <- data.frame(x, y = 1:4)
d_NA[is.na(d_NA)] <- 0
d_NA
#>   x y
#> 1 7 1
#> 2 0 2
#> 3 9 3
#> 4 0 4
```

If we need to do the opposite (insert NA values into a vector) there is a helper function for that. The function is actually `is.na<-` which means we can use it in a somewhat odd way;

```
d_NA <- data.frame(x = 7:10, y = 1:4)
is.na(d_NA$x) <- c(1, 3) ①
d_NA
#>   x y
#> 1 NA 1
```

```
#> 2 8 2
#> 3 NA 3
#> 4 10 4
```

- Specify which values in the vector to set to NA.

Since `is.na()` produces a logical vector, we can use the inverse of this by prepending it with the negation operator `!`, so requesting the non-NA values looks like

```
y <- c(3, 4, NA, NA, 7)
y[!is.na(y)]
#> [1] 3 4 7
```

## 6.6 Selecting Conditionally

All of the above methods of extracting from all of the above structures continue to work if, instead of specifying the rows, columns, or elements we wish to extract, we provide a condition which determines those.

Recall from [Vector Math](#) that automatic recycling of vectors can make comparisons much simpler and more compact

```
test_nums <- c(6, 5, 4, 5, 7, 4)
test_nums > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

Rather than providing a selection of indices to `[` or `[[`, we can instead provide a logical vector of the same length as the row or column we are extracting from, with `TRUE` representing elements we wish to extract, and `FALSE` representing those we don't

```
goodBad <- c("good", "bad", "good", "good", "bad")
goodBad[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
#> [1] "good" "good" "good"
```

Automatic recycling will again take place if the logical vector we supply isn't long enough

```
goodBad[c(TRUE, FALSE)]
#> [1] "good" "good" "bad"
```

and will be partially recycled if its length isn't a multiple of the object's length.

Since vectors themselves can be used to generate these logical vectors, e.g.

```
goodBad == "good"
#> [1] TRUE FALSE TRUE TRUE FALSE
```

we can avoid having to generate this ourselves, and rely only on variable names

```
goodBad[goodBad == "good"]
#> [1] "good" "good" "good"
```

With `data.frames` it's a little more complicated, since we need to refer to the `data.frame` itself to reference columns. For example, to perform the test of whether

the values of `cyl` in the `mtcars` dataset are equal to 6 we can use automatic recycling

```
mtcars$cyl == 6
#> [1] TRUE TRUE FALSE TRUE FALSE TRUE FALSE FALSE FALSE TRUE TRUE
#> [12] FALSE FALSE
#> [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE
# equivalently mtcars[["cyl"]] == 6
```

we could follow the pattern we used in the `goodBad` example above and extract the elements of `mtcars$cyl` which meet this criteria, but that isn't terribly useful. Instead, we can use this logical vector to extract only the *rows* for which the logical vector is TRUE, if we use the `[row, column]` syntax, remembering that leaving one of these options blank retrieves all rows or columns

```
mtcars[mtcars$cyl == 6, ]
#>          mpg cyl disp hp drat    wt  qsec vs am gear carb
#> Mazda RX4   21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
#> Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
#> Valiant     18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> Merc 280    19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
#> Merc 280C   17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
#> Ferrari Dino 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
```

This is the slice of the `mtcars` dataset for which the values in the `cyl` column are 6. The calculation that produces the logical vector can be complicated and involve other columns too. If we wanted to restrict this further to only rows where `am == 1`

```
mtcars[mtcars$cyl == 6 & mtcars$am == 1, ]
#>          mpg cyl disp hp drat    wt  qsec vs am gear carb
#> Mazda RX4   21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
#> Ferrari Dino 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
```

The calculation that builds this logical vector doesn't need to involve the dataset we're slicing either, it can be any calculation that results in a logical vector, such as

```
mtcars[1:8 > 7, ]
#>          mpg cyl disp hp drat    wt  qsec vs am gear carb
#> Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
#> Camaro Z28      13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
#> Volvo 142E      21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

which selects every eighth row, since the first seven values are not greater than 7, and the last is (the resulting vector is recycled sufficiently)

```
1:8 > 7
#> [1] FALSE FALSE FALSE FALSE FALSE FALSE TRUE
```

Using this syntax to select columns works, but as we've said, using the positional index of something is fragile, so I don't recommend this. Nonetheless, we can extract every second column with

```
head(
  mtcars[, c(FALSE, TRUE)]
)
#> cyl  hp   wt vs gear
#> Mazda RX4      6 110 2.620 0    4
#> Mazda RX4 Wag  6 110 2.875 0    4
#> Datsun 710     4  93 2.320 1    4
#> Hornet 4 Drive 6 110 3.215 1    3
#> Hornet Sportabout 8 175 3.440 0    3
#> Valiant        6 105 3.460 1    3
```

If we wish to be a little more robust and select certain column names with a logical vector, we can use the fact that `colnames()` returns a vector. Thus we can create a calculation using this, as complicated as we wish

```
head(
  mtcars[,startsWith(colnames(mtcars), "c") | endsWith(colnames(mtcars), "p")]
)
#> cyl disp  hp carb
#> Mazda RX4      6 160 110 4
#> Mazda RX4 Wag  6 160 110 4
#> Datsun 710     4 108  93 1
#> Hornet 4 Drive 6 258 110 1
#> Hornet Sportabout 8 360 175 2
#> Valiant        6 225 105 1
```

Here I've used the (base) functions `startsWith()` and `endsWith()` we saw in [Text Matching](#) to test whether the `colnames()` of `mtcars` starts/ends with a particular letter. The `|` operator *or* means that the result will be true if either match is found.

If, instead of a full logical vector, we just wish to know *which* values satisfy a condition, we can pass the vector to the `which()` function to produce just the indices of the values which are `TRUE`. For example, the previous example searched the column names of `mtcars` for names starting with `c` using the following

```
startsWith(colnames(mtcars), "c")
#> [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
```

We can turn that into the column *numbers* which match this criteria with

```
which(startsWith(colnames(mtcars), "c"))
#> [1] 2 11
```

Subsetting with `[` can take either a full logical vector or just the indices we wish to keep, so either of these can be used to extract the column names which start with `c`

```
colnames(mtcars)[which(startsWith(colnames(mtcars), "c"))]
#> [1] "cyl"  "carb"
```

All that said, the `dplyr` syntax makes for much cleaner code and achieves the same result (up to inclusion of row names). To select columns whose names start with "`c`" we can use

```
mtcars %>%
```

```
select(starts_with("c")) %>%
  head()
#>          cyl carb
#> Mazda RX4      6    4
#> Mazda RX4 Wag  6    4
#> Datsun 710     4    1
#> Hornet 4 Drive 6    1
#> Hornet Sportabout 8   2
#> Valiant        6    1
```

or to select columns whose name either starts with "c" or ends with "p" we can use a regular expression and the `matches()` helper

```
mtcars %>%
  select(matches("^(c|p$)")) %>% ❶
  head()
#>          cyl disp  hp carb
#> Mazda RX4      6  160 110    4
#> Mazda RX4 Wag  6  160 110    4
#> Datsun 710     4  108  93    1
#> Hornet 4 Drive 6  258 110    1
#> Hornet Sportabout 8  360 175    2
#> Valiant        6  225 105    1
```

❶ The caret ^ matches the start of a string, while the dollar \$ matches the end of a string

## 6.7 Summarising Values

Once we have our subsetted data, we likely wish to use it to calculate some summary value(s). The simplest way to achieve this is the `summary()` function. This is a very useful function; it can work with almost any structure of data and provides an overview summary which is specific to the structure. For a numeric vector, it shows the 5-number summary

```
summary(3:9)
#>   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
#> 3.0    4.5    6.0    6.0    7.5    9.0
```

This details the range of values (minimum and maximum), the mean and median, as well as the first and third quantiles.<sup>53</sup> This shows us how the data are distributed. The mean won't always be equal to the median, in particular if the data is skewed, e.g.

```
set.seed(1)
summary(rlnorm(10))
#>   Min. 1st Qu. Median   Mean 3rd Qu.   Max.
#> 0.4336  0.5851  1.2959  1.5166  1.7409  4.9297
```

When the vector is characters, `summary()` just tells us about the characteristics of the vector

```
summary(letters)
```

<sup>53</sup> If the data was sorted and divided into four equally-sized groups, these would be the values at the first and third internal boundaries. The second boundary defines the median.

```
#>      Length   Class    Mode
#>      26 character character
```

It is a little more helpful when the data is actually **factor** in which case it produces a count of the values in each **factor** level

```
summary(iris$Species)
#>      setosa versicolor virginica
#>      50          50          50
```

For a matrix or a **data.frame** the **summary()** function works over columns and details each of these. For numeric columns it produces the 5-number summary as before. For factor columns it details the counts of each value (if there are only a few)

```
summary(iris[, 3:5])
#>      Petal.Length   Petal.Width       Species
#> Min.   :1.000   Min.   :0.100   setosa   :50
#> 1st Qu.:1.600  1st Qu.:0.300  versicolor:50
#> Median  :4.350  Median  :1.300  virginica:50
#> Mean    :3.758  Mean    :1.199
#> 3rd Qu.:5.100  3rd Qu.:1.800
#> Max.    :6.900  Max.    :2.500
```

This is also useful when there are values missing from your data, as they will be counted below each column's summary. If we have a **data.frame** with missing values, say, an extract of the **mtcars** data

```
mtcars_NA <- mtcars[, 1:4] ①

mtcars_NA[1, 4] <- NA      ②
mtcars_NA[2, 3] <- NA      ②
mtcars_NA[3, 4] <- NA      ②
mtcars_NA[4, 1] <- NA      ②
mtcars_NA[4, 3] <- NA      ②

head(mtcars_NA)
#>           mpg cyl disp  hp
#> Mazda RX4     21.0   6 160 NA
#> Mazda RX4 Wag 21.0   6  NA 110
#> Datsun 710    22.8   4 108 NA
#> Hornet 4 Drive NA    6  NA 110
#> Hornet Sportabout 18.7   8 360 175
#> Valiant        18.1   6 225 105
```

- ① Extracing the first four columns of the **mtcars** data we've been working with, saved to another object for clarity. Even if we modified the **mtcars** object, the original would still be available as **datasets::mtcars**.
- ② Replacing individual values with **NA**.

then the **summary()** of this will let us know how many values are **NA** in each column

```
summary(mtcars_NA)
#>           mpg          cyl          disp          hp
#> Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
```

```
#> 1st Qu.:15.35   1st Qu.:4.000   1st Qu.:120.5   1st Qu.: 99.0
#> Median :19.20   Median :6.000   Median :196.3   Median :136.5
#> Mean    :20.05   Mean    :6.188   Mean    :232.2   Mean    :149.7
#> 3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:342.0   3rd Qu.:180.0
#> Max.    :33.90   Max.    :8.000   Max.    :472.0   Max.    :335.0
#> NA's    :1          NA's    :2          NA's    :2
```

`summary()` also works with more complex structures, and we'll return to these later.

If we wish to distill our data down ourselves, we need another function. The `aggregate()` function from the built-in `stats` package is one way to achieve this for `data.frames`. It takes three main arguments;

- `x`: the `data.frame` to aggregate,
- `by`: a list of grouping elements,
- `FUN`: the function to be used to compute the summary.

In practice, grouping the `iris` dataset (not including the fifth column; the `Species` itself) by the different values of `Species` to calculate the mean of each column looks like

```
aggregate(x = iris[, -5], by = list(Species = iris$Species), mean)
#>      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
#> 1    setosa      5.006     3.428      1.462      0.246
#> 2 versicolor   5.936     2.770      4.260      1.326
#> 3 virginica    6.588     2.974      5.552      2.026
```



### Formula Interface

There is another way to use `aggregate` and that is with a formula. This looks a bit like an equation with a value on the left, a tilde (~) and a value on the right, i.e.

```
y ~ x
```

The way to read this is "y as a function of x". We can use this to define our grouping instead of the list. This looks like

```
aggregate(formula = . ~ Species, data = iris, FUN = mean)
#>      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
#> 1    setosa      5.006     3.428      1.462      0.246
#> 2 versicolor   5.936     2.770      4.260      1.326
#> 3 virginica    6.588     2.974      5.552      2.026
```

Keep in mind that, for whatever reason, the arguments are ordered differently now, so either make sure to use their names or respect the relevant ordering.

This `formula` has a dot (.) which stands for "all columns" on the left, and `Species` (note: not `iris$Species`) on the right, so the grouping is defined to be "all columns as a function of `Species`".

Internally, the components of a `formula` are stored as a list, and the function knows to look for the data in the relevant structure, in the same way as what we saw in ([Non-Standard Evaluation](#)).

I can't say I'm a fan of this function; `dplyr`'s grouped `summarise()` is a much cleaner option.

## 6.8 A Worked Example: Excel vs R

We're finally able to begin working with real data in a meaningful way. It's taken a while, but what we've seen along the way will all come in handy right now.

Let's take the `mtcars` dataset and ask some questions that you might have previously answered using a spreadsheet program. First, what's the average mileage (miles per gallon, `mpg`) of the cars in this data? If you had this data in Excel, you might simply calculate `=AVERAGE(B2:B33)` if `mpg` was the second column. Can you use your new R skills to answer this same question? Create and save a new *script* file and write some code that produces this result. The `mtcars` dataset is already available in your session.

There are of course many ways in which this can be achieved in R. A straightforward way would be to extract the relevant column from the data as a vector and calculate the `mean()` from that

```
mean(mtcars$mpg)
#> [1] 20.09062
```

We could alternatively use the `dplyr` package (which is overkill at this point, but we'll quickly expand to wanting this)

```
library(dplyr)
summarise(mtcars, mileage = mean(mpg))
#>   mileage
#> 1 20.09062
```

Note also that this produces a `data.frame` with just a single row and column. We could convert this to just a number by passing it to `as.numeric()`

```
as.numeric(summarise(mtcars, mileage = mean(mpg)))
#> [1] 20.09062
```

Simple enough.

We could of course use the pipe here too

```
library(dplyr)
mtcars %>%
  summarise(mileage = mean(mpg)) %>%
  as.numeric()
#> [1] 20.09062
```

Our answer using R should match the one we produce using Excel.

Now let's ask another question: what is the mileage for the automatic transmission cars (`am == 0`)? In Excel, you might use a filter to select only those records, and at this point you may notice where the approaches differ. By applying the filter, you have modified the state of your data. Not in a permanent way of course (yet) but you can no longer compare the new average to what you calculated for the full dataset without

undoing the filter.

There are typically two ways around this; creating a copy of the data so that the two calculations can co-exist; or being more explicit in the calculation to perform the filtering in-place, such as with =AVERAGEIF(I2:I33, "=0", B2:B33) (average the values in rows 2 to 33 of the second column if the corresponding value in the ninth column is 0).

The first of these options is unsatisfactory because there is no guarantee that an update to the original data will propagate to the copy. The second option, I would argue, is a programming solution; one that is better suited to a more descriptive programming language. In R, we can add this calculation without modifying or saving unnecessary copies of the data as so

```
mtcars %>%
  filter(am == 0) %>%
  summarise(mileage = mean(mpg)) %>%
  as.numeric()
#> [1] 17.14737
```

If we wish, we can swap back and forth between these two results simply by commenting out the `filter()` step

```
mtcars %>%
#  filter(am == 0) %>% ❶
  summarise(mileage = mean(mpg)) %>%
  as.numeric()
#> [1] 20.09062
```

❶ Note that this line will not be evaluated

so we can assure ourselves that we are indeed comparing what we hope we are.

If we have both of these in our script file, then we can also rest assured that any change to the input data will be reflected in our calculations since we refer to the data with the same variable name (`mtcars` as our example, but this would be whatever you name your data). We can ask and answer as many complex questions as we like, storing the results we wish to keep in new variables, and by writing the code in a script, we preserve the steps which produce those values. Go back to the last Excel file you were working on and ask yourself; "What have I calculated?" Is the filter still active? Is it clear that whatever calculation you've entered into a cell currently represents the answer you were trying to produce? Could you give this to someone and expect them to extract the same answer to your question?

Spreadsheet programs are excellent for displaying tabular data sets and are capable of generating answers to our questions, but they are rarely used in a manner which is reproducible and robust. By scripting our analysis and working with unchanging source data in a descriptive programming language,<sup>54</sup> we create a reproducible flow from

<sup>54</sup> It is worth noting at this point that there is a way to achieve this in Excel; VBA is a programming language that interfaces with Excel.

question to answer.

## 6.9 Try It Yourself

 The `mtcars` data set is available in your session. Try to answer the following questions using both the `base` subsetting functions as well as `dplyr`'s verbs.

Using `mtcars`:

1. What is the maximum (`max()`) weight of the automatic (`am == 1`) vehicles?
2. What is the minimum (`min()`) weight of the manual (`am == 0`) vehicles?
3. What is the mean displacement (`disp`) of 6-cylinder (`cyl`) vehicles?
4. What is the mean displacement (`disp`) of 4-cylinder (`cyl`) vehicles?
5. What is the mean displacement of vehicles grouped by number of cylinders?

Don't peek at the following solutions, but do go back through this chapter to make sure you're comfortable with how to do these.

One way to achieve these results in each of the `base` and `dplyr` styles would be:

1. What is the maximum (`max()`) weight of the automatic (`am == 1`) vehicles?

```
max(mtcars[mtcars$am == 1, "wt"])
#> [1] 3.57
```

```
mtcars %>%
  filter(am == 1) %>%
  summarise(max(wt)) %>%
  as.numeric()
#> [1] 3.57
```

2. What is the minimum (`min()`) weight of the manual (`am == 0`) vehicles?

```
min(mtcars[mtcars$am == 0, "wt"])
#> [1] 2.465
```

```
mtcars %>%
  filter(am == 0) %>%
  summarise(min(wt)) %>%
  as.numeric()
#> [1] 2.465
```

3. What is the mean displacement (`disp`) of 6-cylinder (`cyl`) vehicles?

```
mean(mtcars[mtcars$cyl == 6, "disp"])
#> [1] 183.3143
```

```
mtcars %>%
  filter(cyl == 6) %>%
  summarise(mean(disp)) %>%
  as.numeric()
#> [1] 183.3143
```

4. What is the mean displacement (`disp`) of 4-cylinder (`cyl`) vehicles?

```
mean(mtcars[mtcars$cyl == 4, "disp"])
#> [1] 105.1364

mtcars %>%
  filter(cyl == 4) %>%
  summarise(mean(disp)) %>%
  as.numeric()
#> [1] 105.1364
```

5. What is the mean displacement of vehicles grouped by number of cylinders?

```
aggregate(
  x = mtcars[, "disp", drop = FALSE],
  by = list(cyl = mtcars$cyl),
  FUN = mean
)
#> cyl      disp
#> 1 4 105.1364
#> 2 6 183.3143
#> 3 8 353.1000
```

This one was a bit of a hassle since we still need `x` to be a `data.frame` in order to retain the column name `disp` in the result, and by default R tries to drop this single-column `data.frame` to a vector.

```
mtcars %>%
  group_by(cyl) %>%
  summarise(mean_disp = mean(disp))
#> # A tibble: 3 x 2
#>   cyl  mean_disp
#>   <dbl>
#> 1 4.00    105
#> 2 6.00    183
#> 3 8.00    353
```

## 6.10 Summary

In this chapter we've sliced, diced, and summarised different types of data in several different ways. We've cut bits out and replaced them with other values based on conditions and seen how to distill values down to an overview. The whirlwind tour of `dplyr` and the pipe operator has shown us how the community can extend R's functionality to bring entire new mechanisms into play.

You've learned that:

- You can subset text most easily with functions from the `stringr` package
- Regular expressions make matching text patterns easier
- `[` and `[[` can be used to subset most structures, but with many corner cases
- `[[` selects only a single element
- Subsets can be extracted using `$` for named subsets, e.g. a column of a `data.frame`
- An assignment on a subset replaces the subset, e.g. `vec[2] <- 99`

- The `dplyr` package makes working with `data.frames` easier
- Non-Standard Evaluation changes the way that columns are identified within many functions
- The pipe `%>%` makes chaining operations together easier to read (and write)
- Subsets can be created by conditional selections using logical vectors

New terms you've learned:

#### ***regular expression***

a powerful way of specifying abstract components within text to which a match can be made.

#### ***closure***

proper name for a function, enclosing both an expression and an environment.

#### ***dplyr verbs***

verbs are action words ("run", "eat", "play") and are commonly used to name functions by what action they perform (`mutate`, `select`, `filter`).

#### ***atomic vector***

R's basic vector types (e.g. `logical`, `numeric`, `character`, ...) are the simplest types of vectors. Technically, `lists` are also vectors, but they're not *atomic*, they're *recursive*.

Things to remember:

- Regular expressions (even those not involving special selectors) are case-sensitive.
- The pattern substitution function `sub()` only performs an operation once; to perform it on every match, use `gsub()`.
- Special characters (e.g. `\`) need to be escaped to be used in regular expressions.
- Dollar-name selection will partially match to names by default, but `[[` won't.
- When a `data.frame` subset can be converted to a vector, it will be by default (use `drop = FALSE` to prevent this).
- The `tibble` class `tbl_df` is also a `data.frame` so functions which take a `data.frame` should also be able to take a `tibble`.
- The first argument of `dplyr` verbs is `.data`, but this isn't always true for other functions in other packages, so the pipe (`%>%`) won't work out-of-the-box with `every` function.

We now have the tools we need to start working with **your** data, not just the built-in data sets, so let's see how we can tell R where/what **your** data is and begin working with that.



# I Want To Do Something With Lots of Data

## This chapter covers:

- How to read data into R from different sources
- How to inspect your data once it's in R
- How to write R data to a file

Working with the example data sets has been useful; they're not overly complex yet they have features that help demonstrate the tools we've been learning to use. But they're not **your** data, and that's what you really want to use, so let's explore how to get your data into your workspace, ready to be manipulated/interrogated, and then store it for safe keeping once you're done.

## 7.1 Tidy Data Principles

If you've ever received data from someone else then you've likely encountered the issue of them not having the same mental model of how that data **should** look. Even something as simple as a table of values can be structured in many different ways depending on who created it.

If we have, say, the heights, weights, and ages of a group of people,<sup>55</sup> ([https://en.wikipedia.org/wiki/%C7%83Kung\\_people](https://en.wikipedia.org/wiki/%C7%83Kung_people)), "The data ([Howell1] (<https://github.com/rmcelreath/rethinking/blob/master/data/Howell1.csv>) are partial census data for the Dobe area !Kung San, compiled from interviews conducted by Nancy Howell in the late 1960s.") we might find these stored as a table with repeated labels (one for each person) with a descriptive label for the thing being measured (the *variable*) and a *value* corresponding to that variable. This could be entered as

```
long_df <- data.frame(
  id = c(
    "person1", "person2", "person3", "person1", "person2",
```

---

<sup>55</sup> From Wikipedia

```

"person3", "person1", "person2", "person3", "person1",
"person2", "person3"
),
variable = c(
  "height", "height", "height", "weight", "weight",
  "weight", "age", "age", "age", "male", "male", "male"
),
value = c(151.8, 139.7, 136.5, 47.8, 36.5,
         31.9, 63, 63, 65, 1, 0, 0),
stringsAsFactors = FALSE
)
long_df
#>      id variable value
#> 1 person1    height 151.8
#> 2 person2    height 139.7
#> 3 person3    height 136.5
#> 4 person1    weight  47.8
#> 5 person2    weight  36.5
#> 6 person3    weight  31.9
#> 7 person1      age   63.0
#> 8 person2      age   63.0
#> 9 person3      age   65.0
#> 10 person1     male   1.0
#> 11 person2     male   0.0
#> 12 person3     male   0.0

```

This 'long format' isn't uncommon, and can be useful for reading off pairs of variables and values, or subsetting to certain variables or values. A more usefully structured format however is to use 'tidy' data principles, under which we store data with one row per subject, and one column per observed quantity. In the above example, this means one row per person, and one row per thing being measured (height, weight, age, male/female). This compacts the table down to a 'wide format' where multiple values can be compared across a row

```

wide_df <- data.frame(
  id = c("person1", "person2", "person3"),
  height = c(152, 140, 137),
  weight = c(47.8, 36.5, 31.9),
  age = c(63, 63, 65),
  male = c(1, 0, 0),
  stringsAsFactors = FALSE
)
wide_df
#>      id height weight age male
#> 1 person1    152   47.8  63    1
#> 2 person2    140   36.5  63    0
#> 3 person3    137   31.9  65    0

```

Subsetting can still be performed, but we now have the benefit that we can specify by **which** variables we wish to subset values. This is the structure that we will aim to have our data in, either before or after we receive it. It will also come in handy later when certain package functions expect data in this format. A group of widely used

packages adopts and expects this 'tidy' principle when using the majority of their functions, and these have collectively come to be known as the 'tidyverse'. They are primarily written by the author of `dplyr`'s author Hadley Wickham but the tidyverse has expanded to include a wide variety of packages by many authors and contributors.

Don't worry if your data isn't currently in this format, we'll get to how to go between these two options soon enough. For now, understand the difference and try to classify your data as one or the other.

### 7.1.1 The Working Directory

When we start an R session and assign values to variables in our `workspace`, we are doing so without any specific mention of folders on our computer. The R `workspace` exists only within computer memory, but nonetheless the R session has a connection to the directory structure via the *working directory*. This is where, by default, files will be read from and to, as well as any temporary or ancillary files that R or RStudio creates.

By default, this is the user's home directory (My Documents on Windows, `/home/<username>` on Linux, `/Users/<username>` on Mac, with the cross-platform shortcut `~/`). We can ask R what directory is currently listed as the *working directory* with the `getwd()` (get working directory) function, which takes no arguments

```
getwd() ①
#> [1] "/home/user"
```

- ① The result of this will of course be specific to your computer.

Any files we wish to reference (such as reading from, or saving to) within this *working directory* will be easily referenced by just their name



```
xyz_data <- read.csv("xyzdata.csv") ①
```

- ① In this case, the file `xyzdata.csv` is a file in the current working directory, so `/home/user/xyzdata.csv`.

RStudio will even peek inside sub-folders and help to autocomplete their locations for you. You can still reference files with their full path name, but this requires you to give the *absolute* reference (starting from `~/` or `/` or `C:/` etc...).

If you ever need to share your R code with someone (or even yourself on another computer) then they'll be fairly disappointed to see



```
important_data <- read.csv("D:/Path_to_my_files/important_data.csv")
```

because they won't be able to do anything with your code unless they either change all of the file references, or have their own `D:/Path_to_my_files/` directory.

If you are working within an RStudio project as discussed in [RStudio Projects](#) (and I

suggest you do) then the `here` package will help you manage files relative to that Project's root directory

```
# install.packages("here")
library(here)
#> here() starts at /home/user/myProject
```

This cleverly figures out where your Project is stored and allows you to reference files relative to that directory. This means you don't have to store the full path to your data/files, you can store everything under a Project directory, which means if you share the Project with someone (for example, host it on GitHub) they too will have all the correct file references because `here()` will always refer to that Project's root directory.

If the person you share your code with has a copy of your Project files with the lines



```
library(here)
important_data <- read.csv(
  here("subdirectory_of_project", "important_data.csv"))
```

- ➊ Subdirectories are provided as individual arguments, but equivalently "subdirectory\_of\_project/important\_data.csv".

then they can rest assured that they will know where to look for `important_data.csv`, regardless of where they've saved their copy of your Project.

If we wish to change this *working directory* manually, say to a specific folder in which we will store all the files for an analysis, we can do this with the `setwd()` function (set working directory) which takes a character string describing the location of the folder which you wish to set as the *working directory*



```
setwd("/path/to/my/analysis/folder/")
```

RStudio will help you create this string by autocompleting (press `Tab` part-way through typing a valid folder path).



Depending on which operating system you use, you'll need to take care with the folder path. On Windows systems, the separators between folders are backslashes (\) but as we saw in [Text](#) R treats this as a special character escape code, so we need to use two (escape the backslash with another backslash), i.e. `C:\\\\Users\\\\username`. Alternatively, turn the slashes around, i.e. `C:/Users/username`, which also works.

There is also a more visual way to accomplish this; in RStudio the menu Session → Set Working Directory provides several options including setting the working directory to the current file's location or any other folder you choose.



Be aware that when you start a new session, the working directory may have reverted to a default. Also be aware that other people won't necessarily have access to your computer when you share your script with them, so setting the working directory and referencing data relative to that will mean that they can't reproduce your results. Instead, a better solution is to work with [RStudio Projects](#) (recall [RStudio Projects](#)) in which case all of your directory and file

references can be relative to the project root, made even easier with the `here` package.

### 7.1.2 Stored Data Formats

If you've come from working with spreadsheets, then you're likely most familiar with having your data in a file format relevant to the program you use; `.xls` or `.xlsx` for Excel, `.ods` for Open Office or Libre Office, `.numbers` for Numbers. These formats store more than just your data, they also store the formatting you use, where any plots were located, the formulas, colours, and so on. This used to be a big hurdle for getting the data into R, but things have become a lot easier in the last few years, and this format **can**,<sup>56</sup> be processed by R. These formats have their own issues though, namely that they don't interact well with version control systems (recall [Version Control](#)).

Better suited to version control are plain-text files. These, at their simplest, are just lines of text (letters, numbers, and symbols) in a file with some specific structure to them. This means that a version control system (such as `git`) is able to literally compare two versions of the file and show you what the differences are (if any). One of the most common plain-text data file structures is `.csv` (Comma Separated Values). Data in this format is stored as values separated by commas (shocking, I know). These files can be opened in a text editor, in which case you would see something like



```
id,height,weight,age,male
person1,152,47.8,63,1
person2,140,36.5,63,0
person3,137,31.9,65,0
```

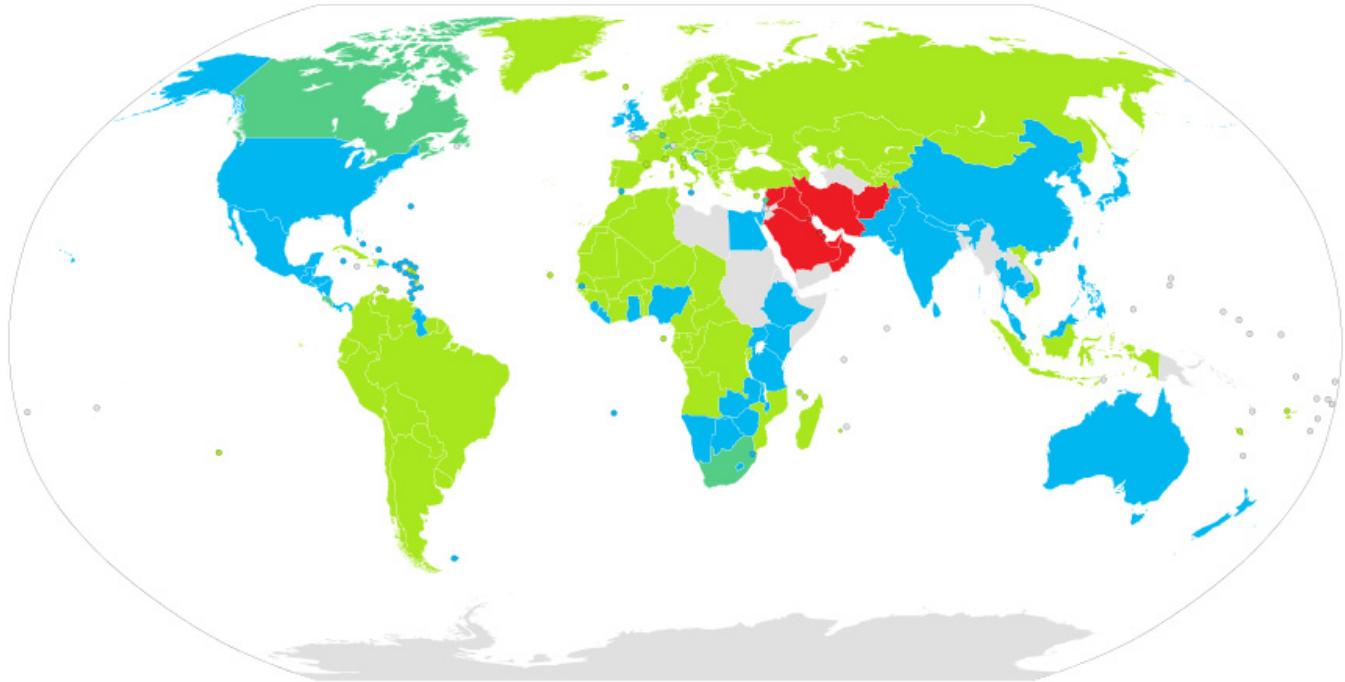
Note that there are no spaces on either side of the commas, just the data values with a comma between them. Spaces are certainly allowed in this format,<sup>57</sup> though great care needs to be taken if any of the text values themselves contain a comma. This can happen if your data happens to be natural language text, such as lines from a book or content from the web. Another likely source of commas is the use of one for denoting decimals. There is little consensus between the use of a period (.) or a comma (,) for decimals, as we can see in Figure 7.1.

---

<sup>56</sup> Under various conditions, mainly that the data is simply structured

<sup>57</sup> There is a proper definition of the CSV format (<https://tools.ietf.org/html/rfc4180>) but this in no way ensures that it is being followed

**Figure 7.1. Countries using . (blue); , (green); and , (red) as their standard decimal separator (those lacking data shown in grey. credit: NuclearVacuum, via Wikimedia Commons.**



A similar problem arises for large numeric values written with a comma as a thousands separator. In this case, it's better to manage the data in a slightly different format.

A straightforward variation on this would be to use a different record delimiter, such as ;, a TAB, a space, or |. Each of these can be dealt with in R, but the same constraints will apply in any case:

- values in fields (records) on a line separated by a delimiter;
- the same number of fields (even if they're blank) on a line;
- lines end with a carriage return (line break);
- no use of the delimiter *within* a record;
- take care with escape characters (\);
- an optional first 'header' line with the names of the fields.

When different delimiters are used, the format is sometimes referred to by that name, such as tab-delimited values (TSV)



id	height	weight	age	male
person1	152	47.8	63	1
person2	140	36.5	63	0
person3	137	31.9	65	0

A more general format forgoes the tabular structure of these and places one record on each line in the file. This could be lines of a poem, names of items in stock, or recordings of a measurement made by a logger. There are fewer restrictions on what this data can contain, but you still need to take care with regards to escape characters.

Text files are the most common way to carefully store data, but they're far from the only way. R can handle almost **any** type of data you happen to have in a file, be that images, GPS-logger files, shapefiles, audio, video, data files from other software packages, databases... just about anything you have.



**There are limitations to this; the packages made to work with those formats may have been written long ago by people no longer maintaining them, they may not deal explicitly with the nuances of your file, and they may require special configuration of your system. This is hardly the fault of R and its packages, whose authors had no say in how those formats were constructed, and in many cases those formats are deliberately made difficult to read by a non-free software company.**

Have a look at the file type of some of your data and identify which it is. Next, we'll work on pointing R to it and reading its contents into memory.

### 7.1.3 Reading Data into R

R has **many** different ways to get data from a stored file into memory, and in most cases there's a function best suited to your particular data structure. For general tabular data the `read.table()` function has 25 configurable options to specify exactly how to read in your data. The first of these, `file`, tells R where to look for the file, and its name (with extension). This is provided as a character string either with the full directory path, or relative to the *working directory*. The returned value of this function, if successful, is a `data.frame` representing the contents of the file, and by default this would simply be printed to the `Console` unless we save it to a variable



```
myTabularData <- read.table(file = "myFile.dat")
```



This function does not set up a *connection* to the contents of that file, it merely opens the file, reads the values into memory, then closes the file. Any changes you make to either the file or the variable containing the data after this point will not be reflected in the other, unless we explicitly read/write again.

The remaining options are set to reasonable defaults, but may not meet your specific needs. If we have CSV data, we would need to change the `sep = ""` (blank) option to `sep = ","`. Thankfully, there are convenience functions for several common formats, namely `read.csv()` and `read.delim()` which internally call `read.table()` but have more specific defaults set.

If it's feasible and makes sense to do so (data in a single table) then converting a proprietary format (e.g. `.xlsx`) to CSV can make reading the data into R a lot simpler. It's important though to remember that once you make a copy of the data, the link back

to the original is broken, so any changes you make in either won't be in sync.

These functions aren't limited to working with local files; the `read.table()` help documentation for the `file` argument notes that this can also be a URL (web link), so we can pull in data from a remote site

```
Howell_data <-  
"https://github.com/rmcelreath/rethinking/raw/master/data/Howell1.csv"  
  
head(  
  read.table(  
    file = Howell_data, ②  
    sep = ";", ③  
    header = TRUE ④  
  )  
)  
#>   height  weight age male  
#> 1 151.765 47.82561 63   1  
#> 2 139.700 36.48581 63   0  
#> 3 136.525 31.86484 65   0  
#> 4 156.845 53.04191 41   1  
#> 5 145.415 41.27687 51   0  
#> 6 163.830 62.99259 35   1
```

- ① Just printing the first six rows to preview.
- ② Using the string stored in the variable `Howell_data` as the `file` argument value.
- ③ This file uses ; as a record separator.
- ④ The column names are provided on the first row, so we use this option.

This can be very useful for several reasons; you may wish to store the offsite location of the data and reference it in your *script* to increase reproducibility (you won't need to share the data yourself, or the source may be more trustworthy than an e-mailed data file). You may be working on data that is updated by someone else and hosted, and storing that data may not be necessary for your analysis.

For example, if you were processing rainfall data provided by your local city's weather bureau, and wanted to produce a plot of the daily rainfall for the year, you might be satisfied to download the entire series of values each day instead of trying to only download a single day of values and maintain your own dataset (which transfers the responsibility of data assurance to you, and which may be more trouble in any case).



**It's quite common for spreadsheets to have column names with spaces, punctuation, start with numbers... all the things that standard `data.frame` column names can't have (without special treatment). Recall from `data.frame`'s that spaces and punctuation will be converted to dots, while columns starting with a number (or dot-number) will be renamed to start with an X.**

If the values are a simple table and are stored in a spreadsheet program such as Excel and you're using R on Windows,<sup>58</sup> then there's a neat way to copy a table of data (or

---

<sup>58</sup> as far as I can tell, integration of this feature on Mac and Linux isn't great.

subset of one) into R without saving the file into a new permanent file.<sup>59</sup> The `file` argument to `read.table()` can, as we've seen, refer to a local file or a URL, but it can also be the keyword "clipboard" in which case any data you've most recently 'copied' from a spreadsheet or table-like text will be available and an attempt will be made to read it in to memory. If we have some cells in a spreadsheet, such as those shown in Figure 7.2, we can select and copy them to the system clipboard (e.g. File → Copy or `ctrl + c` depending on the system/program)

	A	B	C	D	E
1	height	weight	age	male	
2	151.765	47.82561	63	1	
3	139.7	36.48581	63	0	
4	136.525	31.86484	65	0	
5	156.845	53.04191	41	1	
6	145.415	41.27687	51	0	
7	163.83	62.99259	35	1	
8					
9					

**Figure 7.1. A selection of cells in a spreadsheet.**

at which point we can use `read.table()` to read in this copied information



```
## NOTE this may only work on Windows
read.table("clipboard", header = TRUE)
#> height weight age male
#> 1 151.765 47.82561 63 1
#> 2 139.700 36.48581 63 0
#> 3 136.525 31.86484 65 0
#> 4 156.845 53.04191 41 1
#> 5 145.415 41.27687 51 0
#> 6 163.830 62.99259 35 1
```



A package which automates this process more generally, and also provides a way to 'paste' in the code that creates an object, is the `datapasta` package, available on CRAN. This does support most platforms, so it also works on Mac and Linux.

When we don't wish to save a different version of an Excel file, perhaps because it has several sheets which belong together, then there are options to connect directly to `.xls` and `.xlsx` files. Several packages have been written over the years to make this easier, but the one I recommend most strongly is the `rio` package.

```
# install.packages("rio")
library(rio)
```

This is able to import more than thirty different file formats with a consistent syntax; `import()` is able to read-in a wide variety of file formats

---

<sup>59</sup> Don't worry if you can't find much information about this; it's seemingly undocumented.



```
excelContents <- import(file = "myExcelDocument.xls")
```

and can optionally select a specific Excel sheet from which to import a table.

If your data is in a less common format then it's a good idea to research whether or not R is already able to read it, otherwise search for packages which provide this support. There are very few file formats (beyond uncommon laboratory equipment which uses its own file format) which have zero support in R.

One of the largest annoyances people find with importing data into R is the assumptions made over what *type* the imported data is in. A CSV file doesn't store this information along with the data, so a row of data that looks like



```
3,apple,7.4,2,banana
```

will be read in treating all of the numbers as class *numeric* (not *integer*) and (because the default of `stringsAsFactors` is `TRUE`) the text values as class *factor*. Be careful over whether or not this is the result you wish to have, and set `stringsAsFactors = FALSE` when importing data if it's not. The `import()` function from the `rio` package sets this automatically.

If your data is simply lines to be read one at a time into values, then the `readLines()` function may be of use. This function, when the `con` (connection) argument is provided as a character string, opens the file of that name (if it can) and reads the requested number of lines (by default, all of them) in to the R session as a character *vector*. For example, if we have a file in the *working directory* named `myFile.txt` with the following contents



```
line one
line two
line three
```

then we can read this in to R with



```
readLines("myFile.txt")
#> [1] "Line one"    "Line two"    "Line three"
```



`readLines()` will read the lines in as a *character vector*, regardless of the *type* of the data. If your data is not text, you will need to manually convert the result of `readLines()`.

There are many other ways in which you can connect R to your data source, such as connecting to databases (e.g. MySQL, SQLite, or Access), reading in a specific filetype (there are myriad packages which offer some sort of `read*`() function), or simply connecting to a remote source. This last option warrants its own discussion as this can be lead to great sources of data.

## 7.1.4 Scraping Data

### Web Scraping

Content from the World Wide Web can be *scraped* (copied, collected) by using software to automatically extract the data from a webpage's source file rather than simply viewing it.

Sometimes you don't have your own source for data, but it is hosted somewhere else as a table on a webpage and you wish to use the values for an analysis. You could copy out the table, format it, and save it to a file, but that involves a lot of manual processing that you would have to repeat if the data changed or you wanted to repeat the analysis.

In this case, *scraping* the data may be a possibility.<sup>60</sup> If it is permitted, then we can tell R where to look on a webpage to read in some data. The `rvest` package provides useful functions to extract information out of the underlying HTML of a webpage.

```
# install.packages("rvest")
library(rvest)
# Loading required package: xml2
```

### HTML

Hypertext Markup Language (HTML) is how webpages are constructed; using tags to change the appearance and layout of different elements. If you're not familiar at all with HTML but need to perform web scraping then it would be a good idea to do some reading up.

Walking through an example might make this clearest; let's scrape the table of milestones for R releases from Wikipedia;

[https://en.wikipedia.org/wiki/R\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/R_(programming_language)).

The table itself is rendered by a browser as shown in Figure 7. 3.

---

<sup>60</sup> Not all sites allow this, and it is highly advisable to check the usage policy of a site before attempting this.

**Figure 7.2. Table of milestones for R releases. Captured from [https://en.wikipedia.org/wiki/R\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/R_(programming_language)) Jan 2018.**

## Milestones [edit]

A list of changes in R releases is maintained in various "news" files at CRAN.<sup>[43]</sup> Some highlights are listed below for several major releases.

Release	Date	Description
0.16		This is the last <a href="#">alpha</a> version developed primarily by Ihaka and Gentleman. Much of the basic functionality from the "White Book" (see <a href="#">S history</a> ) was implemented. The mailing lists commenced on April 1, 1997.
0.49	1997-04-23	This is the oldest <a href="#">source</a> release which is currently available on CRAN. <sup>[44]</sup> CRAN is started on this date, with 3 mirrors that initially hosted 12 packages. <sup>[45]</sup> Alpha versions of R for Microsoft Windows and the <a href="#">classic Mac OS</a> are made available shortly after this version. <sup>[citation needed]</sup>
0.60	1997-12-05	R becomes an official part of the <a href="#">GNU Project</a> . The code is hosted and maintained on <a href="#">CVS</a> .
0.65.1	1999-10-07	First versions of update.packages and install.packages functions for downloading and installing packages from CRAN. <sup>[46]</sup>
1.0	2000-02-29	Considered by its developers stable enough for production use. <sup>[47]</sup>
1.4	2001-12-19	S4 methods are introduced and the first version for <a href="#">Mac OS X</a> is made available soon after.
2.0	2004-10-04	Introduced <a href="#">lazy loading</a> , which enables fast loading of data with minimal expense of system memory.
2.1	2005-04-18	Support for <a href="#">UTF-8</a> encoding, and the beginnings of <a href="#">internationalization and localization</a> for different languages.
2.11	2010-04-22	Support for Windows 64 bit systems.
2.13	2011-04-14	Adding a new compiler function that allows speeding up functions by converting them to byte-code.
2.14	2011-10-31	Added mandatory namespaces for packages. Added a new parallel package.
2.15	2012-03-30	New load balancing functions. Improved serialization speed for long vectors.
3.0	2013-04-03	Support for numeric index values $2^{31}$ and larger on 64 bit systems.

We can save the URL (Uniform Resource Locator, a.k.a. web address) for the Wikipedia page as a character string which saves typing it repeatedly

```
r_lang_URL <- "https://en.wikipedia.org/wiki/R_(programming_language)"
```

We can extract the HTML contents of the page into a special object with the `read_html()` function from `rvest`

```
r_html <- read_html(r_lang_URL)
```

This opens a connection to the Wikipedia server and extracts the raw code which your browser would otherwise interpret for you and present on a page. This won't look like it makes much sense if you look at the contents, but we'll process it a little and get out the bits we actually want. Don't be worried if you get a Warning along the lines of 'closing unused connection...' at any point; it's just `R` cleaning up connections it no longer needs.

From this, we can extract all elements which have the HTML tag `table`

```
html_nodes(r_html, "table")
#> [xml_nodeset (9)]
#> [1] <table class="infobox event" style="width:22em">\n#> [2] <table class="wikitable">\n\nRelease\nDate\n<"" span="" style="box-sizing: border-box;">
#> [3] <table class="plainlinks metadata ambox ambox-content ambox-Weasel" ...
#> [4] <table class="plainlinks metadata ambox ambox-content ambox-peacock" ...
#> [5] <table class="nowraplinks collapsible autocollapse navbox-inner" sty ...
#> [6] <table class="nowraplinks collapsible autocollapse navbox-inner" sty ...
#> [7] <table class="nowraplinks hlist collapsible collapsed navbox-inner" ...
#> [8] <table class="nowraplinks navbox-subgroup" style="border-spacing:0"> ...
```

```
#> [9] <table class="nowraplinks hlist navbox-inner" style="border-spacing: ...
```

The structure of these extractions is:

- `x` to select a tag element (`<x>`, e.g. `<table>`)
- `.y` to select a `y` class (`<x class="y">`, e.g. `<div class="foo">`)
- `#z` to select a named tag (`<x id="z">`, e.g. `<p id="foo">`)

If we 'View Source' for the Wikipedia page in a browser and search for 'table' we see that the 'Milestones' table is a table of class 'wikitable': `<table class="wikitable">` we can select this part of the source with `html_node()` (since there is only one wikitable on the page)

```
release_html <- html_node(r_html, ".wikitable")
```

Recalling that a period prepending a class is the selector we need here. The object returned from `html_node()` can be converted to a `data.frame`, at which point we have successfully read a table from a webpage into a usable R structure

```
release_table <- html_table(release_html)
```

Printing the first two columns (excluding the descriptions) we can verify that we have correctly extracted the table (comparing to the rendered result in Figure 7.3.).

```
release_table[, 1:2]
#>   Release      Date
#> 1 0.16
#> 2 0.49 1997-04-23
#> 3 0.60 1997-12-05
#> 4 0.65.1 1999-10-07
#> 5 1.0 2000-02-29
#> 6 1.4 2001-12-19
#> 7 2.0 2004-10-04
#> 8 2.1 2005-04-18
#> 9 2.11 2010-04-22
#> 10 2.13 2011-04-14
#> 11 2.14 2011-10-31
#> 12 2.15 2012-03-30
#> 13 3.0 2013-04-03
```

We could also extract the table of contents (toc) using the `toc` tagged node (recalling that `#z` selects the tag named `z`)

```
tag_toc_html <- html_nodes(r_html, "#toc")
```

We can then extract the links of the unordered list (`<ul>`) which each have a list item (`<li>`) tag and an anchor (`<a>`) tag

```
toplevel_toc_links <- html_nodes(tag_toc_html, "ul li a")
```

Finally, we can extract the text from these links and view the result

```
toplevel_toc_text <- html_text(toplevel_toc_links)
toplevel_toc_text
#> [1] "1 History"
```

```
#> [2] "2 Statistical features"
#> [3] "3 Programming features"
#> [4] "4 Packages"
#> [5] "5 Milestones"
#> [6] "6 Interfaces"
#> [7] "7 Implementations"
#> [8] "8 user! conferences"
#> [9] "9 R Journal"
#> [10] "10 Comparison with SAS, SPSS, and Stata"
#> [11] "11 Commercial support for R"
#> [12] "12 Examples"
#> [13] "12.1 Basic syntax"
#> [14] "12.2 Structure of a function"
#> [15] "12.3 Mandelbrot set"
#> [16] "13 See also"
#> [17] "14 References"
#> [18] "15 External Links"
```

The above walkthrough involved saving many intermediate variables, which we likely won't need again. As we've seen, there is a better way to manage this pattern, and that's using chaining with `%>%` from the `magrittr` (or if we're already using it, `dplyr`) package. The `rvest` package actually makes `%>%` available also, so we're ready to start

```
r_lang_URL %>%
  read_html() %>%
  html_node(".wikitable") %>%
  html_table() %>%
  .[c(1, 2)] # select the first two columns
#>   Release      Date
#> 1    0.16
#> 2    0.49 1997-04-23
#> 3    0.60 1997-12-05
#> 4    0.65.1 1999-10-07
#> 5    1.0 2000-02-29
#> 6    1.4 2001-12-19
#> 7    2.0 2004-10-04
#> 8    2.1 2005-04-18
#> 9    2.11 2010-04-22
#> 10   2.13 2011-04-14
#> 11   2.14 2011-10-31
#> 12   2.15 2012-03-30
#> 13   3.0 2013-04-03
```

The table of contents scraping steps can be distilled down to

```
r_lang_URL %>%
  read_html() %>%
  html_nodes("#toc") %>%
  html_nodes("ul li a") %>%
  html_text # extract the Link text
#> [1] "1 History"
#> [2] "2 Statistical features"
#> [3] "3 Programming features"
#> [4] "4 Packages"
```

```
#> [5] "5 Milestones"
#> [6] "6 Interfaces"
#> [7] "7 Implementations"
#> [8] "8 user! conferences"
#> [9] "9 R Journal"
#> [10] "10 Comparison with SAS, SPSS, and Stata"
#> [11] "11 Commercial support for R"
#> [12] "12 Examples"
#> [13] "12.1 Basic syntax"
#> [14] "12.2 Structure of a function"
#> [15] "12.3 Mandelbrot set"
#> [16] "13 See also"
#> [17] "14 References"
#> [18] "15 External Links"
```

Clearly the pipe makes these steps much simpler and cleaner, so I strongly recommend building up your scraping pipelines in this way. Add each step sequentially, checking that you are producing the structure you expect along the way.

### 7.1.5 Inspecting Data

We've already seen the `summary()` function which gives us an overview of an object. When we have *continuous* data these 5-number summaries are useful for picturing how the data is distributed. When we have *discrete* data however, things are a little different. If a column is characters then we don't get very useful summary. When a column is a `factor`, `summary()` gives us a count of the values in each level.

Another way to generate this which *does* work for character types is the `table()` function. When used on a `factor` type it produces the same output as `summary()` (except it shows **all** levels, not just the first few)

```
table(iris$Species) ①
#>
#>     setosa versicolor virginica
#>     50      50      50
```

① The `Species` column of the `iris` dataset is of class `factor`.

On a `character` type, it produces the same; a count of values in each unique 'level'

```
table(as.character(iris$Species))
#>
#>     setosa versicolor virginica
#>     50      50      50
```

Note that this will respect any differences such as upper and lowercase, so `aBc` and `Abc` are counted as 'different'

```
table(c("aBc", "Abc", "aBc", "aBc", "Abc"))
#>
#> aBc Abc
#> 3    2
```

When used on a numeric vector, the results become overwhelming as each different

value is considered as its own 'level'

```
table(mtcars$mpg)
#>
#> 10.4 13.3 14.3 14.7 15 15.2 15.5 15.8 16.4 17.3 17.8 18.1 18.7 19.2 19.7
#> 2 1 1 1 1 2 1 1 1 1 1 1 1 1 2 1 1
#> 21 21.4 21.5 22.8 24.4 26 27.3 30.4 32.4 33.9
#> 2 2 1 2 1 1 1 2 1 1
```

When there are only a few unique values, this can still be very handy

```
table(mtcars$carb)
#>
#> 1 2 3 4 6 8
#> 7 10 3 10 1 1
```

Sometimes it's just as useful to look at the top, bottom, or random values to see what's going on.

The `head()` function (which you've seen already throughout the earlier chapters) prints the first (by default, six) rows of a `data.frame` (or similar, e.g. `matrix`, `tibble`, `table`) object, which is handy for checking the column names along with some of the data. It can be useful to check that there are actual values you expect to see in those rows (e.g. not all NA, not all 0, not all strings). For example,

```
head(iris)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1 5.1 3.5 1.4 0.2 setosa
#> 2 4.9 3.0 1.4 0.2 setosa
#> 3 4.7 3.2 1.3 0.2 setosa
#> 4 4.6 3.1 1.5 0.2 setosa
#> 5 5.0 3.6 1.4 0.2 setosa
#> 6 5.4 3.9 1.7 0.4 setosa
```

Similarly, the `tail()` function prints the lasts (by default, six) rows

```
tail(iris)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 145 6.7 3.3 5.7 2.5 virginica
#> 146 6.7 3.0 5.2 2.3 virginica
#> 147 6.3 2.5 5.0 1.9 virginica
#> 148 6.5 3.0 5.2 2.0 virginica
#> 149 6.2 3.4 5.4 2.3 virginica
#> 150 5.9 3.0 5.1 1.8 virginica
```

The number of rows to be printed can be controlled with the `n` argument, if you require more or fewer. If you want more of a peek throughout the data you could randomly sample the rows and print those. There is a `dplyr` function which makes this easy

```
set.seed(1) 1
dplyr::sample_n(mtcars, 6)
#>          mpg cyl disp hp drat wt qsec vs am gear carb
#> Merc 230 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4 2
#> Merc 450SE 16.4 8 275.8 180 3.07 4.070 17.40 0 0 3 3
#> Fiat 128 32.4 4 78.7 66 4.08 2.200 19.47 1 1 4 1
```

```
#> Porsche 914-2    26.0   4 120.3  91 4.43 2.140 16.70  0  1     5     2
#> Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0     3     1
#> Pontiac Firebird 19.2   8 400.0 175 3.08 3.845 17.05  0  0     3     2
```

- ➊ Setting the seed value ensures that you too will get this same random sample.

If you wish to view an object in its entirety then the `View()` function (note the capital V in `View()`) is very handy. When passed a data argument, this opens a data viewer (in the Editor pane, so it has its own file-like tab) containing the data, converted to a `data.frame` (where possible). This works in both a terminal and RStudio, though the latter has better support for filtering, sorting, and displaying. Give it a try; `View(mtcars)` should show something like Figure 7.4.

**Figure 7.3. Data Viewer invoked with `View()`**

	mpg	cyl	disp	hp	drat	wt	qsec
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02
Valiant	18.1	6	225.0	105	2.76	3.460	20.22
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00

### 7.1.6 I Have Odd Values In My Data (Sentinel Values)

#### Sentinel value

Also known as flag, trip, rogue, or signal values, these are used in data entry to highlight certain records. Perhaps an impossibly high or low value will be pushed to 99 or -99, or a missing value will be encoded as 0 or -1.

We've already covered having `NA` values in your data, but this isn't the only value you need to be on the lookout for. Sentinel values typically arise as shortcuts in data entry. They attempt to bring attention to themselves by standing out distinct from 'regular' values. In records of local daily temperature, you may find some values of 999 which probably aren't an indication that the measuring station was on fire, but rather the value encodes some information that is hopefully recorded elsewhere, such as noting that the equipment was failing or the measurement somehow invalid.

This is an okay practice, as long as everyone in contact with the data is clear about what the values represent, and have the appropriate key to decipher this code.

Spotting these values is trivial if you know what they are ahead of time. Let's say we were working on quality-control of the `mtcars` data and wanted to highlight that the

horsepower (`hp`) of the Maserati Bora seemed excessively high, or wasn't measured correctly, or couldn't be trusted for whatever reason. It's not uncommon to have a placeholder value which sticks out. In this case, the processing might look like

```
mtcars_qc <- mtcars
mtcars_qc["Maserati Bora", "hp"] ①
#> [1] 335
mtcars_qc["Maserati Bora", "hp"] <- 999 ②
tail(mtcars_qc) ③
#>          mpg cyl disp hp drat    wt qsec vs am gear carb
#> Porsche 914-2 26.0   4 120.3 91 4.43 2.140 16.7  0  1    5    2
#> Lotus Europa  30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
#> Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
#> Ferrari Dino  19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
#> Maserati Bora  15.0   8 301.0 999 3.54 3.570 14.6  0  1    5    8
#> Volvo 142E    21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

- ③ Replacing the `hp` value for this row with a sentinel value.
- ④ `tail()` prints the *last* six rows (recall `head()` prints the *first* six).

Given that we know how to subset data conditionally, we can look for known values specifically and select them. An example search might look like

```
mtcars_qc[mtcars_qc$hp == 999, ] ①
#>          mpg cyl disp hp drat    wt qsec vs am gear carb
#> Maserati Bora 15     8 301 999 3.54 3.57 14.6  0  1    5    8
```

- ① Using the base square-bracket method which preserves row names, whereas `dplyr::filter()` does not.

If we found some and wanted to remove those records, we could do so similarly

```
tail(mtcars_qc[mtcars_qc$hp != 999, ])
#>          mpg cyl disp hp drat    wt qsec vs am gear carb
#> Fiat X1-9   27.3   4 79.0 66 4.08 1.935 18.9  1  1    4    1
#> Porsche 914-2 26.0   4 120.3 91 4.43 2.140 16.7  0  1    5    2
#> Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
#> Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
#> Ferrari Dino 19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
#> Volvo 142E   21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

We have more work to do when we aren't sure that any of these values exist in our data, but we can be diligent and check. A good sentinel value will be obvious in the context of the remaining data, so one way to check for these is to examine the distribution of values. We've seen how to do that already using the `summary()` function. From the summary, it can be easy to see that a Min. or Max. value stands out from the rest

```
summary(mtcars_qc[, 2:5]) ①
#>      cyl           disp          hp          drat
#> Min. :4.000   Min. : 71.1   Min. : 52.0   Min. :2.760
#> 1st Qu.:4.000  1st Qu.:120.8  1st Qu.: 96.5  1st Qu.:3.080
```

```
#> Median :6.000  Median :196.3  Median :123.0  Median :3.695
#> Mean   :6.188  Mean   :230.7  Mean   :167.4  Mean   :3.597
#> 3rd Qu.:8.000 3rd Qu.:326.0 3rd Qu.:180.0 3rd Qu.:3.920
#> Max.   :8.000  Max.   :472.0  Max.   :999.0  Max.   :4.930
```

- ① Just checking columns two through five.

Note that the `Max.` value of `hp` is *much* larger than the `Mean` value.

`NA` itself is a sentinel value in that it encodes missingness. Other possible sentinel values include infinity (`Inf` or `-Inf`) which can arise when one divides some number by 0; or `NaN` which tends to indicate  $0/0$ . Each of these can be captured by the `is.finite()` function which returns `FALSE` for `NA`, `Inf`, `-Inf`, and `NaN`. If we combine this with the `any()` function (which returns `TRUE` if *any* of the values passed to it are `TRUE`) we can test for all of these at once

```
any(!is.finite(c(1, 2, 3, 4)))
#> [1] FALSE

any(!is.finite(c(1, 2, NA, 4)))
#> [1] TRUE

any(!is.finite(c(1, 2, NaN, 4)))
#> [1] TRUE

any(!is.finite(c(1, 2, Inf, 4)))
#> [1] TRUE
```

### 7.1.7 Converting to Tidy Data

Once you have read your data into R, it's time to do something with it. Often, the data isn't in the structure that an analysis function requires. Most of the 'tidyverse' expects the 'wide' structure; one row per observation and one column per observable/measurable quantity. If you find that your data is in the 'long' format and you need it to be wide, this can be most easily achieved with the `tidyr` package

```
# install.packages("tidyverse")
library(tidyverse)
```

we can use the `spread()` function to spread the data out into a wide format. If we begin with the long-format data from the start of this chapter

```
long_df
#>      id variable value
#> 1 person1  height 151.8
#> 2 person2  height 139.7
#> 3 person3  height 136.5
#> 4 person1  weight  47.8
#> 5 person2  weight  36.5
#> 6 person3  weight  31.9
#> 7 person1    age   63.0
#> 8 person2    age   63.0
#> 9 person3    age   65.0
#> 10 person1   male   1.0
#> 11 person2   male   0.0
```

```
#> 12 person3     male    0.0
```

then use `spread()` to create new columns using the variable column as the key to new column names, with values from the value column

```
wide_df <- spread(data = long_df, key = variable, value = value)
wide_df
#>      id age height male weight
#> 1 person1 63 151.8    1   47.8
#> 2 person2 63 139.7    0   36.5
#> 3 person3 65 136.5    0   31.9
```

This is now in the 'wide' format we require.

If we *do* need to go the other way, the `gather()` function from `tidyverse` gathers up a selection of columns and makes a 'long' format object

```
long_df_again <- gather(
  data = wide_df,
  key = "variable", ❶
  value = "value", ❷
  -id ❸
)
long_df_again
#>      id variable value
#> 1 person1     age 63.0
#> 2 person2     age 63.0
#> 3 person3     age 65.0
#> 4 person1     height 151.8
#> 5 person2     height 139.7
#> 6 person3     height 136.5
#> 7 person1     male   1.0
#> 8 person2     male   0.0
#> 9 person3     male   0.0
#> 10 person1    weight 47.8
#> 11 person2    weight 36.5
#> 12 person3    weight 31.9
```

- ❶ The name we wish to give to the column of variables.
- ❷ The name we wish to give to the column of values.
- ❸ Selecting which columns to gather, in this case all *except* the `id` column.



### A base way to reshape data

There's good reason to begin with this 'tidyverse' way of chaining from 'long' to 'wide' data and one of those reasons is that the base way of doing so is confusing and difficult. The built-in `stats` package does have a `reshape()` function but it's not nearly as elegant:

```
reshape(
  long_df,
  idvar = "id",
  timevar = "variable",
  direction = "wide"
)
```

```
#>      id value.height value.weight value.age value.male
#> 1 person1      151.8        47.8       63         1
#> 2 person2      139.7        36.5       63         0
#> 3 person3      136.5        31.9       65         0
```

Notice that the column names have the prefix value.. The argument names are complex and difficult to recall.

The reshape2 package makes this significantly easier. If we load that package

```
# install.packages("reshape2")
library(reshape2) ①
#
#> Attaching package: 'reshape2'
#> The following object is masked from 'package:tidyR':
#>
#>     smiths
```

- ① The tidyR and reshape2 packages both contain the smiths dataset; the former encoded as a tibble, the latter as a data.frame. Loading and attaching the reshape2 package *after* the tidyR package *masks* which version will become available if you were to enter smiths into the Console.

we can use the dcast() function which performs the same task as above using a formula interface

```
dcast(long_df, id ~ variable)
#>      id age height male weight
#> 1 person1 63 151.8   1  47.8
#> 2 person2 63 139.7   0  36.5
#> 3 person3 65 136.5   0  31.9
```

tidyR is built as a replacement to reshape2, so it's not surprising that it works similarly easily.

Other things to check include:

- Do you have/want characters or factors?

Should you use tibbles or set the argument stringsAsFactors?

- What are the levels/unique values of your columns?

Are there more/fewer than you expect? Do they correctly represent your data? Are there obvious typos?

- Do you have any missing values? Sentinel values?

Do you expect any columns to have NAs? What about the range of present values?

- Do you have repeated rows/observations?

The last of these — repeated rows — occurs regularly. R will occasionally warn you when this is the case, but not always. To determine if you have duplicate rows you can use the duplicated() function from the base package. If we have a data.frame with repeated measurements, say

```
rep_df <- data.frame(
  name = c("x", "y", "y", "z", "x"),
```

```

    val1 = c(3, 5, 5, 4, 8),
    val2 = c(9, 2, 2, 1, 1),
    val3 = c("q", "r", "r", "s", "t")
)
rep_df
#>   name val1 val2 val3
#> 1   x     3     9     q
#> 2   y     5     2     r
#> 3   y     5     2     r
#> 4   z     4     1     s
#> 5   x     8     1     t

```

we can ask "which rows are duplicates" (appear more than once) with

```

duplicated(rep_df)
#> [1] FALSE FALSE  TRUE FALSE FALSE

```

This returns a logical vector corresponding to which rows have already appeared, reading sequentially from row 1. Since row 2 has not appeared previously, it is not — at that point — a duplicate. Row 3 however is the second appearance of these values, so it is a duplicate. Rows 1 and 5 both have "x" in the `name` column but the other values make the row unique. If we reverse the search (from the last row) we see the same issue, in that there is only one duplicate

```

duplicated(rep_df, fromLast = TRUE)
#> [1] FALSE  TRUE FALSE FALSE FALSE

```

We can therefore remove this duplicate row since we have a logical vector which we can negate to identify which rows we wish to keep

```

rep_df[!duplicated(rep_df), ]
#>   name val1 val2 val3
#> 1   x     3     9     q
#> 2   y     5     2     r
#> 4   z     4     1     s
#> 5   x     8     1     t

```

Alternatively, there is a shortcut to this exact expression

```

unique(rep_df)
#>   name val1 val2 val3
#> 1   x     3     9     q
#> 2   y     5     2     r
#> 4   z     4     1     s
#> 5   x     8     1     t

```

This function also works on vectors, so we can also extract unique values from these

```

unique(rep_df$val2)
#> [1] 9 2 1

```

## 7.2 Merging Data

Having individual pieces of data stored in individual `data.frames` or `tibbles` makes each of them easy to access and work with, but there are times when we need to bring

two of these objects together. This most often happens when we have some overlap (say, a column of identifiers) between two which link each record. The separate objects may be consistent on their own and represent different things (this *is* a good way to store data) but if we need to compare between the two properties then we need a way to join the objects.

If we have, say, a **tibble** representing some countries and their capitals (which perhaps contain some data specific to our analysis)

```
# install.packages("tibble")
library(tibble)
countries_capitals <- tribble( ❶
  ~Country,      ~Capital,
  "Australia",   "Canberra",
  "France",      "Paris",
  "Malaysia",    "Kuala Lumpur",
  "India",        "New Delhi",
  "Nauru",        NA, ❷
  "Russia",       "Moscow",
  "Switzerland",  NA, ❷
  "Zimbabwe",    "Harare"
)
```

- ❶ We can create a **tibble** row-by-row with **tribble**.
- ❷ Nauru and Switzerland have no official capitals, only *de facto* capitals.

and another larger **tibble** representing the populations of capital cities around the world sourced from elsewhere (in this case, [https://en.wikipedia.org/wiki/List\\_of\\_national\\_capitals\\_by\\_population](https://en.wikipedia.org/wiki/List_of_national_capitals_by_population) as of the end of 2017)

```
cities_populations <- tribble(
  ~City, ~Population,
  "New Delhi",  16787949L,
  "Moscow",     11541000L,
  "Paris",      2241346L,
  "Harare",     1487028L,
  "Kuala Lumpur", 1381830L,
  "Dublin",     1173179L,
  "Ottawa",     898150L,
  "Wellington", 405000L,
  "Canberra",   354644L
)
```

then we can join these using one of **dplyr**'s join verbs. These borrow from the syntax of SQL<sup>61</sup> and perform merging operations.

Of course, if we haven't already, we need to install, load, and attach the **dplyr** library

```
# install.packages("dplyr")
library(dplyr)
```

---

<sup>61</sup> Structured Query Language, a language built for interacting with databases.

```

#
#> Attaching package: 'dplyr'
# The following object is masked from 'package:switchr':
#>
#>     location
# The following objects are masked from 'package:stats':
#>
#>     filter, lag
# The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

```

The most straightforward option is to perform a 'full join' of the two objects; take all of the records from the first and match up with all of the records from the second, joining on the column representing the capital city, ensuring that any missing records from either are kept

```

full_join(
  countries_capitals,          1
  cities_populations,          2
  by = c("Capital" = "City")  3
)
#> # A tibble: 11 x 3
#>   Country     Capital     Population
#>   <chr>
#> 1 Australia   Canberra    354644
#> 2 France      Paris       2241346
#> 3 Malaysia    Kuala Lumpur 1381830
#> 4 India       New Delhi   16787949
#> 5 Nauru       <NA>        NA
#> 6 Russia      Moscow      11541000
#> 7 Switzerland <NA>        NA
#> 8 Zimbabwe    Harare     1487028
#> 9 <NA>         Dublin     1173179
#> 10 <NA>        Ottawa     898150
#> 11 <NA>        Wellington 405000

```

- ① The first object, containing the countries and their capitals.
- ② The second object, containing the cities (capitals) and their populations.
- ③ Here we indicate that the columns which have the common information are the Capital and City columns. We will join the two objects 'by' this column. If the column name(s) are common between the two objects, this doesn't need to be specified, otherwise the linkage is spelled out as we have here.

Notice that every row from both the first and second objects is now present; where there was no information to merge in, the element is `NA`. Nauru and Switzerland have no official capitals so their population columns remain as `NA`. Similarly, Our `tibble` of countries and capitals didn't include Canada, Ireland, or New Zealand, so the capitals of those (Ottawa, Dublin, Wellington, respectively) have `NA`s as their country.



### Important

When joining/merging two objects, only the information found in the row(s) containing the common value(s) between them will appear in the resulting object. The common value(s) must exactly (according to R) match for this to work. If we try the above example with some not-quite-matching common values

```
dataA <- tribble(
  ~name, ~value,
  "x", 1,
  "y", 2,
  "z", 3
)
dataB <- tribble(
  ~name, ~other_value,
  "x", 4,
  "Y", 5, 1
  "z", 6
)
full_join(dataA, dataB)
# Joining, by = "name"
#> # A tibble: 4 x 3
#>   name  value other_value
#>   <chr> <dbl>     <dbl>
#> 1 x      1.00      4.00
#> 2 y      2.00      NA
#> 3 z      3.00      6.00
#> 4 Y      NA        5.00
```

- 1 Note that in this object, the second name is uppercase, whereas in the first object it is lowercase. This therefore isn't a match and both y and Y will have rows in the object produced using a `full_join()`.

Joining two objects with a text string as the common value linking them is very common, but do take care over uniqueness of them. If a common value appears more than once, it will also be repeated in the resulting object

```
dataA <- tribble(
  ~name, ~value,
  "x", 1.1,
  "x", 1.2,
  "y", 2,
  "z", 3
)
dataB <- tribble(
  ~name, ~other_value,
  "x", 4,
  "y", 5,
  "z", 6
)
full_join(dataA, dataB)
# Joining, by = "name"
#> # A tibble: 4 x 3
#>   name  value other_value
#>   <chr> <dbl>     <dbl>
#> 1 x      1.10      4.00
#> 2 x      1.20      NA
#> 3 y      2.00      5.00
#> 4 z      3.00      6.00
```

```
#> 2 x      1.20      4.00
#> 3 y      2.00      5.00
#> 4 z      3.00      6.00
```

Having non-unique common values becomes particularly important when we remember how R treats numbers; are `2L` and `2` the same in this context? As far as `dplyr` is concerned, they are, so you can in theory have numeric or integer values as the common link between two columns. Just be very cautious about how exact those values are and check for uniqueness in the resulting object once you're done.

So far we've seen the scenario where we want all of the rows of the first object and all of the rows of the second, but what about when one object contains more information than we require? Returning to our population example, what if we *only* want the populations of capitals in our original object?

`dplyr` also has `left_join()` for situations like this; the resulting object will contain all of the rows and columns of the first (in a one-line call, the object on the left) object, and all of the columns of the second object where there was a matching common value

```
left_join(
  countries_capitals,
  cities_populations,
  by = c("Capital" = "City")
)
#> # A tibble: 8 x 3
#>   Country     Capital     Population
#>   <chr>
#> 1 Australia  Canberra    354644
#> 2 France     Paris       2241346
#> 3 Malaysia   Kuala Lumpur 1381830
#> 4 India      New Delhi   16787949
#> 5 Nauru      <NA>        NA
#> 6 Russia     Moscow      11541000
#> 7 Switzerland <NA>        NA
#> 8 Zimbabwe   Harare     1487028
```

Note here that this new object does not contain the rows which were only present in the `cities_populations` object, but all of those from `countries_capitals` (8 rows).

The complement to this is `right_join()`; the resulting object will contain all of the rows and columns of the second object, and all of the columns of the first object where there was a matching common value

```
right_join(
  countries_capitals,
  cities_populations,
  by = c("Capital" = "City")
)
#> # A tibble: 9 x 3
#>   Country     Capital     Population
#>   <chr>
#> 1 India      New Delhi   16787949
#> 2 Russia     Moscow      11541000
```

```
#> 3 France    Paris        2241346
#> 4 Zimbabwe Harare      1487028
#> 5 Malaysia  Kuala Lumpur 1381830
#> 6 <NA>      Dublin      1173179
#> 7 <NA>      Ottawa      898150
#> 8 <NA>      Wellington 405000
#> 9 Australia Canberra   354644
```

Here, this new object does not contain the rows which were only present in the `countries_capitals` object, but all of those from `cities_populations` (9 rows).

An `inner_join()` produces an object which contains only the overlapping rows (does not add rows of missing columns). This is very handy when you want to only work with the data you actually have values for

```
inner_join(
  countries_capitals,
  cities_populations,
  by = c("Capital" = "City")
)
#> # A tibble: 6 x 3
#>   Country     Capital     Population
#>   <chr>
#> 1 Australia  Canberra    354644
#> 2 France     Paris       2241346
#> 3 Malaysia   Kuala Lumpur 1381830
#> 4 India      New Delhi   16787949
#> 5 Russia     Moscow      11541000
#> 6 Zimbabwe   Harare     1487028
```

Finally, there is a way to join using only rows which appear in one object but *not* in another, `anti_join()`. This is especially useful for determining which rows are missing from some object

```
anti_join(
  countries_capitals,
  cities_populations,
  by = c("Capital" = "City")
)
#> # A tibble: 2 x 2
#>   Country     Capital
#>   <chr>
#> 1 Nauru      <NA>
#> 2 Switzerland <NA>
```

This produces an object with the rows from `countries_capitals` which don't appear in `cities_populations`; these countries have no `Capital` value so they have no `City` value on which to match. If we want the complement to this we need to swap the first two arguments and rearrange the `by` argument

```
anti_join(
  cities_populations,
  countries_capitals,
  by = c("City" = "Capital")
```

```
)
#> # A tibble: 3 x 2
#>   City      Population
#>   <chr>
#> 1 Dublin     1173179
#> 2 Ottawa      898150
#> 3 Wellington 405000
```

This produces an object with the rows appearing in `cities_populations` but missing from `countries_capitals`.



### A base way to merge data

base R has a way to merge data.frames too; the `merge()` function works similarly to `inner_join()` (with some slightly less convenient quirks)

```
merge(
  countries_capitals, ❶
  cities_populations,
  by.x = "Capital", ❷
  by.y = "City"       ❷
)
#>      Capital  Country Population
#> 1    Canberra Australia     354644
#> 2      Harare Zimbabwe    1487028
#> 3  Kuala Lumpur Malaysia   1381830
#> 4      Moscow  Russia  11541000
#> 5    New Delhi   India  16787949
#> 6      Paris    France  2241346
```

- ❶ Note that since tibbles also have class `data.frame`, this works without any additional conversion.
- ❷ The structure of the `by` argument is more specific in `merge()`.

If we want something like `full_join()`, we can specify the `all` argument

```
merge(
  countries_capitals,
  cities_populations,
  by.x = "Capital",
  by.y = "City",
  all = TRUE
)
#>      Capital  Country Population
#> 1    Canberra Australia     354644
#> 2      Dublin        <NA>  1173179
#> 3      Harare Zimbabwe    1487028
#> 4  Kuala Lumpur Malaysia   1381830
#> 5      Moscow  Russia  11541000
#> 6    New Delhi   India  16787949
#> 7      Ottawa        <NA>  898150
#> 8      Paris    France  2241346
#> 9    Wellington        <NA>  405000
#> 10      <NA>      Nauru       NA
#> 11      <NA> Switzerland     NA
```

If we want something like `left_join()`, we can specify just the `all.x` argument

```
merge(
  countries_capitals,
  cities_populations,
  by.x = "Capital",
  by.y = "City",
  all.x = TRUE
)
#>      Capital      Country Population
#> 1    Canberra    Australia     354644
#> 2      Harare    Zimbabwe    1487028
#> 3   Kuala Lumpur Malaysia    1381830
#> 4      Moscow     Russia    11541000
#> 5   New Delhi     India    16787949
#> 6      Paris      France    2241346
#> 7      <NA>      Nauru       NA
#> 8      <NA> Switzerland     NA
```

The biggest advantages of dplyr's join functions over the base `::merge()` are that it preserves the row order (`merge()` tends to move them around) and is substantially (potentially several fold) faster (since it's written in C under the hood).

## 7.3 Writing Data From R

After we've finished investigating our data (or even during that process) we will eventually need to save our modified/cleaned/summarised data back to the disk. All of the data we've been working with has so far only existed within the ephemeral memory of the R session.

There are several ways we can save our data, and just as with reading data *into* R, which one to use largely depends on your particular use-case.

The simplest way to save an R object to disk is with the `save()` function. This takes (via the `...` argument) some number of names of objects (as either bare names or character strings) you wish to save (or a list of them via the `list` argument) and a filename (the `file` argument, optionally including the full path relative to the working directory, and typically ending with `.RData`). You can specify multiple objects in the `...` or `list` argument, and these will all be saved within the one `.RData` file



```
z <- 3
save(mtcars, z, iris, file = "example_data_YYMMDD.RData")
```

This creates the file `example_data_YYMMDD.RData` in your working directory (or wherever you specify).



**Remember to use sensible names for your files so that you will know what they represent later... `mydata.RData` isn't particularly informative.**

The major benefit of this method of saving data is that the data is saved preserving the R structure, not converted to text or flattened into a table. If you have a `tibble` with a list column and you save it using `save()`, then reading it back in will restore that exact object. Speaking of reading back in, a `.RData` file can be loaded back into a new session using the `load()` function which takes a filename as the `file` argument



```
load(file = "example_data_YYMMDD.RData")
```

If this is evaluated in a fresh, clean, empty R session, the Environment pane will now show the variables `mtcars`, `z`, and `iris`.

We can save *all* of our variables this way too. Listing each and every object we've created would be much too tedious, so thankfully there's a shortcut for this; `save.image()`. If that function sounds familiar, it may be because all the way back in Chapter 1 ([Working with R within RStudio](#)) I advised you to turn *off* the default option of "*save your workspace image*". This still stands; it's poor practice to automatically save your workspace, but there certainly may be times you wish to do it on purpose.

By default, `save.image()` saves to a file named `.RData` in your working directory (the filename beginning with a dot means it is treated as a hidden file in many operating systems) but you can name that whatever you like with the `file` argument. This image is created using `save()`, so it is loaded just the same as other `.RData` files; using `load()`.

If you only have a single object to save, an alternative is to use `saveRDS` which behaves slightly differently. A single object can be specified using the `object` argument, and a filename (typically ending with `.rds`, and optionally including the relative path) instructs where to save the file. In this case, I recommend using the object name in the name of the file, to assist with recalling what is saved, since the original object name won't be saved in this case



```
fruits <- c("strawberries", "apples", "watermelon")
saveRDS(fruits, file = "red_fruits.rds")
```

These files have a slight advantage over `.RData` files as they are compressed, so they take up less disk space.

Loading these in is slightly different; we use the `readRDS()` function which takes a `file` argument and instead of *loading* the object into the workspace, it *returns* it



```
readRDS(file = "red_fruits.rds")
#> [1] "strawberries" "apples"           "watermelon"
```

By itself, this just prints the object to the `Console`, but more usefully, it can be stored in a variable or used in further processing



```
fruit_salad <- c("banana", "grapes", readRDS(file = "red_fruits.rds"))
fruit_salad
#> [1] "banana"       "grapes"        "strawberries" "apples"        "watermelon"
```

As you can see, the name of the new variable need not be the same as the one used when saving the object (or necessarily used at all) which can be both a blessing (when you need to save multiple copies of an object to files then read them in) and a curse (when you fail to name the file correctly and can't recall what data it represents).

One last structure-preserving save-mechanism: If you're planning on opening a Stack Overflow question because your code isn't working the way you expect it to, you'll often be (rightfully) asked to include some data which allows others to reproduce the issue you're facing. In this case, you want others to be able to take some of your data, apply your code, and dig into what's going wrong. If they don't have the right data, they can't be expected to encounter the same issue you do. In this case, sharing a saved file can be a bit difficult, so there's another way we can output a structured object.

The `dput()` function writes out some R code that would generate the object you pass to it. If we have a `data.frame` with a few rows and columns of different types

```
mixed_type_df <- data.frame(
  ints   = 1:4,                      1
  nums   = 2.5:5.5,                  2
  facts  = factor(letters[23:26]),    3
  chars  = state.abb[1:4],           4
  stringsAsFactors = FALSE          3 4
)
mixed_type_df
#>   ints nums facts chars
#> 1   1   2.5     w   AL
#> 2   2   3.5     x   AK
#> 3   3   4.5     y   AZ
#> 4   4   5.5     z   AR
```

- ➊ With this formatting, the sequence will be integers.
- ➋ With this formatting, the sequence will be numeric.
- ➌ If we wish to have *some* factor columns, despite `stringsAsFactors=FALSE`, we need to specify them explicitly.
- ➍ `state.abb` is a built-in dataset of the abbreviations of USA state names. These will remain as characters due to `stringsAsFactors=FALSE`.

We can see from the structure that the columns have different types, so we wish to preserve that

```
str(mixed_type_df)
#> 'data.frame':      4 obs. of  4 variables:
#> $ ints : int  1 2 3 4
#> $ nums : num  2.5 3.5 4.5 5.5
#> $ facts: Factor w/ 4 Levels "w","x","y","z": 1 2 3 4
#> $ chars: chr  "AL" "AK" "AZ" "AR"
```

For simple examples, sharing *that* code would be simplest, but this is only simple code because I've relied on sequences. The `dput()` function writes some R code to generate this structure

```
dput(mixed_type_df)
#> structure(list(ints = 1:4, nums = c(2.5, 3.5, 4.5, 5.5), facts = structure(1:4,
#> .Label = c("w",
#> "x", "y", "z"), class = "factor"), chars = c("AL", "AK", "AZ",
#> "AR")), .Names = c("ints", "nums", "facts", "chars"), row.names = c(NA,
#> -4L), class = "data.frame")
```

This looks like a mess, but it contains all R needs to know to re-build `mixed_type_df`. If someone else wanted to reproduce that object, they can assign that code to `mixed_type_df` and know they're working with the same example data. Breaking up the lines into something readable and storing the result in a variable, it looks like

```
mixed_type_df <- structure(
  list(ints = 1:4,
       nums = c(2.5, 3.5, 4.5, 5.5),
       facts = structure(
         1:4,
         .Label = c("w", "x", "y", "z"),
         class = "factor"
       ),
       chars = c("AL", "AK", "AZ", "AR")
     ),
     .Names = c("ints", "nums", "facts", "chars"),
     row.names = c(NA, -4L), class = "data.frame"
   )
mixed_type_df
#>   ints nums facts chars
#> 1    1  2.5    w    AL
#> 2    2  3.5    x    AK
#> 3    3  4.5    y    AZ
#> 4    4  5.5    z    AR
```

If you feel that the output from `dput()` is too long, it's probably a sign that your example data is too large. Remember, if you're trying to highlight a particular issue you're facing, try paring it down to just the part of your data that seems to be involved in the issue. An extra twenty columns that are uninvolved in your analysis just get in the way of the question you're asking.

When we have tabular data and are likely to want/need to open it in a spreadsheet program (or other software which requires such) it can be useful to write the data back to that format. We saw how to read in tabular data (e.g. `.csv`) files earlier, and sure enough there's a counterpart to that; `write.table()` and the convenience function `write.csv()`.

These take a single data object as an input and write a relevant file to a filename specified in the `file` argument (optionally, with relative path). This is very handy for tabular structures such as `matrix` and `data.frame`. We could write the `mtcars` data set to a `.csv` file with

```
write.csv(mtcars, file = "mtcars.csv")
```

By default, this writes the `row.names` to the file also. In the case of `mtcars` this has benefit (since the names of the cars are stored there) but in general this will be a bit bothersome, so I usually turn it off with the argument `row.names = FALSE`.

There are other more complex ways to save data too; it is possible to write to Excel workbooks (`.xls/.xlsx`) though this often requires jumping through a few hoops. There are optimised methods for saving *large* data objects.<sup>62</sup> There are ways to communicate with databases.<sup>63</sup> There are of course many ways to write images to disk, but we'll cover those when we discuss plotting.

## 7.4 Try It Yourself

 Try scraping the table of Australian City Populations: [https://en.wikipedia.org/wiki/List\\_of\\_cities\\_in\\_Australia\\_by\\_population](https://en.wikipedia.org/wiki/List_of_cities_in_Australia_by_population)

Hints: the table has class `wikitable` so it can be selected in the same way as the previous examples.

## 7.5 Summary

In this chapter we've seen many ways we can read and write our data to and from R, be that in comma separated values, the clipboard, general text, or a table on a webpage. We've learned more about 'tidy' data and how to inspect our data.

You've learned that:

- Long and wide data structures are different ways to store the same data
- R has a the working directory where it will start looking for (or try to store) files
- Most plain text stored data can be read in by R
- R can read data from external sources, such as online
- Web pages can be scraped directly into R objects using the `rvest` package
- The `summary()` function can help reveal the presence of sentinel values
- You can join two `data.frames` or `tibbles` with `dplyr`
- Different join functions include different overlaps between tables
- Multiple objects can be stored in a `.RData` file
- Single objects can be efficiently stored in a `.rds` file

New terms you've learned:

### ***working directory***

the directory in which R will look for files when reading/writing. Set with the `setwd()` function, and queried with the `getwd()` function.

<sup>62</sup> <https://blog.rstudio.org/2016/03/29/feather/>

<sup>63</sup> Actually, `dplyr` handles this quite well, abstracting away most of the database interaction.

**(web) scraping**

collecting data from an online resource using software to process the raw HTML source.

**sentinel value**

a blatantly incorrect value in a dataset chosen to draw attention to some other meaning.

**long data**

when single observations are broken up between several rows, involving the name of the observed measurement as an entry, we regard this as 'long data'.

**wide data**

when a single row corresponds to a single observation, with values spread across columns denoting measured quantities, we regard this as 'wide data', and this conforms to the 'tidy' principle.

Things to remember:

- You can specify file locations relative to an RStudio Project with the `here` package, or set the working directory yourself with `setwd()`.
- Path names in Windows use a backslash; make sure to escape this in strings referring to paths, or turn them around.
- Be aware of the separator used in your plain-text data files.
- Duplicated values in `duplicated()` only show up on their *second* occurrence.
- The `base` function `merge()` sorts rows alphabetically, whereas `dplyr`'s `*_join()` leave them as they were.
- When writing out to a file, `row.names=FALSE` will prevent writing of `row.names` values.
- When saving objects, `save()` stores the name and contents of an object, and is retrieved with `load()`. Using `saveRDS()` instead does not keep the name, so this needs to be assigned (if you want it) after retrieving with `readRDS()`.

You should now have enough tools to get started and actually get some data into R, inspect it, slice it up, modify it, and save it back to a file. I encourage you to go and do that before moving on further. Find some data, get it into R, and play around with it. My guess is that within the first half hour of doing this (provided you're able to) you'll ask at least one of two questions: "how do I do this repeatedly" or "how do I do this conditionally"?

Once you're satisfied that you understand what you're doing up to this point, we'll move on to answering those questions with control structures.

# I Want To Do Something Conditionally (Control Structures)

## This chapter covers:

- Looping over sections of code
- Conditionally evaluating code
- Best practices to balance speed and readability

We've seen that we can conditionally select elements from data structures in [Selecting Conditionally](#) but sometimes we only want to do (or not do) something based on some condition, or do something many times with slightly different conditions. It may be many steps which we wish to conditionally perform, so we need a way to only do things sometimes (or many times). There are several ways to do this in R, so let's have a look at the most common and the most useful, as well as some other ways these tasks can be accomplished.

## 8.1 Looping

So far we've evaluated code in a very linear manner; one line after another in a series of steps. This works fine as long as each step is a unique operation. Often though we wish to repeat an operation several times (sometimes a few, sometimes a large number of times). This comes up most frequently when people wish to perform some operation on each row of a `data.frame` one at a time, but that usage flies in the face of the vectorised nature of R, and should really be discouraged.

We'll still go through how to construct code in that way, but you'll greatly benefit by thinking 'vectorisation' whenever you feel the urge to write some code that performs a *loop*. To return that notion to the front of your mind, let's recall what we learned in [Vector Math Operations](#).

### 8.1.1 Vectorisation

We have already seen that, because R naturally uses vectors (operators know how to

combine values from vectors, and *recycle* values to a common length) we can use operators between two vectors (even if one only has a single element) and produce a vector output. The example we used was

```
c(6, 5, 4, 5, 7, 4) > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

where R recycles the smaller vector (5) to the same length as the longer vector (which it does whenever it can) then performs the requested operation (>; the 'greater than?' comparison) on each element-pair individually. Each comparison results in a TRUE or FALSE value, and together these are combined in a new vector.

Because most of these vectorised operations are performed at the C level rather than the R level,<sup>64</sup> they are typically heavily optimised and complete very quickly.

If we had these values stored in a specific vector

```
test_nums <- c(6, 5, 4, 5, 7, 4)
```

Then we could (but shouldn't) write out each of the comparisons between each one-element subset and the value 5 ourselves and store all of the results in a vector

```
is_test_nums_gt_5 <- c(
  test_nums[1] > 5,
  test_nums[2] > 5,
  test_nums[3] > 5,
  test_nums[4] > 5,
  test_nums[5] > 5,
  test_nums[6] > 5
)
is_test_nums_gt_5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

Whenever you see code that seems to be repeating itself over and over it's usually a sign that you can perform some vectorisation. Too often though it's taken as a sign that you need a loop over the repeated parts. We can compact the repetition down by treating the part that changes as a sequence (and/or a vector), so we can perform all of the comparisons using

```
test_nums[1:6] > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

We can of course take full advantage of the vectorised nature of the operator (the function `>`) and request that the entire vectors be compared this way

```
test_nums > 5
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

There is a minor advantage to the explicit form in that we may wish to alter the order in which these are compared.

We might recall from our `areaCircle()` example in [Functions](#) that this pattern lends

<sup>64</sup> Recall that many of the internal functions in R are actually compiled C or Fortran code.

itself to *abstraction*; replacing values with some arbitrary placeholder and allowing it to change. This is the basis of *looping* over code in R; where the placeholder value is repeatedly changed for us.

There is a construct in R that does exactly this ("for each of these placeholder values, do this...") but I strongly advise you to learn a more sophisticated way before you go down that path (which leads to poor practices and a nightmare for debugging). With that in mind, it's time to introduce the `purrr` package.

### 8.1.2 Tidy Repetition (Looping with `purrr`)

The `purrr` package (be sure to include the three r's) is another package I almost always load at the start of an analysis. Rather than just vectorising a single operation, we can think of just about everything as a series of functions to apply repeatedly to elements.



#### Make your functions `purrr`

**Why three r's?** Legend has it the package's author Hadley Wickham liked the 'pure(r)' sound of this name and wanted the same number of characters as `dplyr` and other tidyverse-style packages; <https://github.com/tidyverse/purrr/issues/35>. The more R the better, I say.

If we have installed the `purrr` package using `install.packages("purrr")` and loaded/attached it using

```
# install.packages("purrr")
library(purrr)
```

then what once might have been long and complicated repetition code is now easy to write.

The basic concept we'll be focussing on here is the `map()` behaviour. In our chapter on Functions ([Understanding the Tools We'll Use — Functions](#)) we had a construct which took in some argument and performed some operation on it. `map()` is a similar idea, but now we'll provide both the inputs and the function to which they will be passed, one at a time.

The object returned by `map()` will be a *list*, and it will *always* be a list.<sup>65</sup> A simple example of this is to use an existing function, say, `sqrt()` which takes the square root of its input

```
sqrt(9)
#> [1] 3
```

This is a trivial example, because as we've seen, `sqrt()` can take a vector as its argument and returns a vector of square roots. This won't always be the case; as we'll see, the function we're working with can be arbitrarily complicated and won't necessarily work as intended with a vector argument.

If we wish to pass several values to this function one at a time (effectively *loop* over a vector of values) then we can do so with `map()` by providing the 'list' of values (an

---

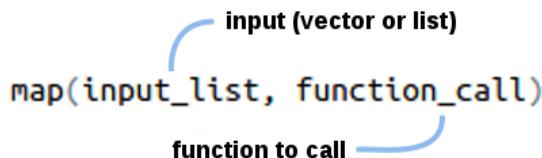
<sup>65</sup> We'll see why this is an important point when we discuss the constructs that `map()` replaces

actual `list` or a `vector`) and the function to which they should be input

```
map(c(4, 9, 16, 25), sqrt)
#> [[1]]
#> [1] 2
#>
#> [[2]]
#> [1] 3
#>
#> [[3]]
#> [1] 4
#>
#> [[4]]
#> [1] 5
```

The construction of calls to this function and the inputs are described in Figure 8. 1.

**Figure 8.1. purrr `map()` construction**



As with all of the 'tidy' functions, the data we are using must be the first argument (formally named `.x` here).<sup>66</sup> The function argument (formally named `.f`) is also provided, and one might note that it is not surrounded by quotes. This means that we are passing the actual function object itself as an argument (R is perfectly okay with this, as we saw in [Multiple Arguments](#)). Additional arguments to the function can also be provided using the ... mechanism (any arguments other than `.x` or `.f` will be passed along to the specified function when called).

Now, a `list` as the return object might not be exactly what you want (though sometimes it might). As an added bonus, there are more specific `map_*` functions which have specific (*consistent*) return types. If instead of a `list` we wanted to return a numeric vector, we can do so with `map_dbl()` ("map to double") which is called in the same way as above

```
map_dbl(c(4, 9, 16, 25), sqrt)
#> [1] 2 3 4 5
```

If, instead, we wanted to return a character vector, it's simply a matter of calling `map_chr()` ("map to character")

```
map_chr(c(4, 9, 16, 25), sqrt)
#> [1] "2.000000" "3.000000" "4.000000" "5.000000"
```

Note in this last example that the result is a character vector, created by first

<sup>66</sup> This means that the pipe operator `%>%` will also work well with this function

calculating the square roots of the values mapped over, then converting the (numeric) results to characters.

This becomes most powerful when the function you are applying to each list input returns a `data.frame` but you want the overall return type to be a single `data.frame`. If we select two rows at random from the `mtcars` dataset using `dplyr`'s `sample_n()` function (with a specified `seed` value so you can get the exact same results on your machine)

```
set.seed(2)
dplyr::sample_n(tbl = mtcars, size = 2)
#>          mpg cyl disp hp drat    wt  qsec vs am gear carb
#> Valiant     18.1   6 225 105 2.76 3.46 20.22  1  0    3    1
#> Dodge Challenger 15.5   8 318 150 2.76 3.52 16.87  0  0    3    2
```

and we wished to repeat this three times for three values of `size`, we could simply use the ordinary `map()`

```
set.seed(2)
map(c(2, 3, 2), dplyr::sample_n, tbl = mtcars) ①
#> [[1]]
#>          mpg cyl disp hp drat    wt  qsec vs am gear carb
#> Valiant     18.1   6 225 105 2.76 3.46 20.22  1  0    3    1
#> Dodge Challenger 15.5   8 318 150 2.76 3.52 16.87  0  0    3    2
#>
#> [[2]]
#>          mpg cyl disp hp drat    wt  qsec vs am gear carb
#> Honda Civic  30.4   4 75.7 52 4.93 1.615 18.52  1  1    4    2
#> Valiant     18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
#>
#> [[3]]
#>          mpg cyl disp hp drat    wt  qsec vs am gear carb
#> Maserati Bora 15.0   8 301 335 3.54 3.57 14.60  0  1    5    8
#> Hornet Sportabout 18.7   8 360 175 3.15 3.44 17.02  0  0    3    2
```

- ① The `tbl` argument to `dplyr::sample_n()` is provided via the `...` mechanism, while the `size` argument is *looped* over.

If we wanted just a `data.frame` as the result, we can use `map_df()` to join these outputs together, binding their rows on top of each other

```
set.seed(2)
map_df(c(2, 3, 2), dplyr::sample_n, tbl = mtcars)
#>          mpg cyl disp hp drat    wt  qsec vs am gear carb
#> 1 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> 2 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
#> 3 30.4   4 75.7 52 4.93 1.615 18.52  1  1    4    2
#> 4 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> 5 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
#> 6 15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
#> 7 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
```



Note that the row names were not output in the `map_df()` example. As a reminder; This is not uncommon in the 'tidyverse' of packages, but it can lead to surprise if you're expecting it to work. The use of row names to store data is highly discouraged since it disconnects the metadata from the actual data. If these names are important, they should be in a column of their own. If you find yourself wanting to enforce that rule onto a `data.frame` with `row.names` then the `tibble` package has several functions for interacting with these, such as `rownames_to_column()`.

A slightly different version of this is the `walk()` function which is useful when the function you are mapping over produces a *side effect* rather than returns a result. This is often producing a plot or printing something to the Console, where you don't actually want anything returned. For example, if we have a function which should only print a result

```
print_value <- function(x) {
  print(paste("The value of x is ", x))
}
```

and we wanted to loop over this, we *could* use `map()`

```
ans_map <- map(1:3, print_value)
#> [1] "The value of x is 1"
#> [1] "The value of x is 2"
#> [1] "The value of x is 3"
```

```
ans_map
#> [[1]]
#> [1] "The value of x is 1"
#>
#> [[2]]
#> [1] "The value of x is 2"
#>
#> [[3]]
#> [1] "The value of x is 3"
```

but this returns the result into the variable. If we weren't assigning it, it too would `print()`. Instead, we can discard this returned data and use `walk()`

```
walk(1:3, print_value)
#> [1] "The value of x is 1"
#> [1] "The value of x is 2"
#> [1] "The value of x is 3"
```

`walk()` actually returns the input in case we wish to do more with it, so

```
ans_walk <- walk(1:3, print_value)
#> [1] "The value of x is 1"
#> [1] "The value of x is 2"
#> [1] "The value of x is 3"

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

The function argument to `map()` (and related functions) is very flexible; instead of an existing function, we can also provide a *formula* which describes an unnamed

(*anonymous*, sometimes called *lambda*) function.

### **Anonymous Function**

A function which is not assigned to a variable name. In some R functions these can have a shorthand formula notation such as `~ .x + 1` to mean `function(.x) .x + 1`.

If we simply wanted to add 1 to our input list, we could do so without necessarily creating an `add_one()` function

```
map_dbl(c(4, 5, 6), ~ .x + 1) ①
#> [1] 5 6 7
```

- ① The `formula ~ .x + 1` is equivalent to `function(x) x + 1`.

We can of course write our own functions and refer to these by name. If we replace the 'compare `test_nums` to 5' operation with a function taking a general argument, we need a function which has a single argument (the index of the element to which the vector will be subset), so we could write

```
compare_to_five <- function(value) { ① ②
  return(value > 5)
} ③
```

- ① The name of the argument is arbitrary, so here we're using `value` to refer to the value being compared to 5.
- ② The name of the function should reflect its action using a verb. In this case the value being compared is fixed so it is also included in the name.
- ③ An explicit call to `return()` isn't necessary, but it doesn't hurt to include it.

This is a fairly trivial function, but it's of great benefit to think this way. If we wish to provide several inputs to this function (all of the values in `test_nums`) returning a logical vector of the results, then that is straightforward

```
map_lgl(test_nums, compare_to_five)
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

of course, since the `>` operator already accepts vector input, so does our function (by extension)

```
compare_to_five(test_nums)
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

Again, with more complicated inputs and functions, this won't always be the case.



The family of `map` functions have older ancestors in base R known as the `apply` family. These are `apply()`, `sapply()`, `lapply()` and some others. They have the same goals as the `map` family but they lack **type safety** in that the `typeof` object returned is not guaranteed to be consistent with different inputs, which leads to some unexpected results. Nonetheless, these show up often in published code so it's a good idea to recognise them.

For example, the `sapply()` function takes a vector and a function, and (most of the time)

**returns a vector of the results**

```
sapply(test_nums, compare_to_five)
#> [1] TRUE FALSE FALSE FALSE TRUE FALSE
```

Danger arises because `sapply()` can actually return several different types of object, depending on the input. base R does have a type-safe version, `vapply()` but this requires you to specify the type each time you want to use it. You won't go wrong if you stick with the `map` family.

While I strongly recommend you learn how to use these `purrr` functions, there **will** be times when you'll see more explicit (and discouraged) looping, so we'll also cover how that works next.

### 8.1.3 `for` loops

In the above example we explicitly created a sequence of indices which identified which values to compare to the number 5. If we replace the sequence 1, 2, 3, ..., 6 with a placeholder, say `i` for index



```
test_nums[i] > 5
```

then we can perform this operation for each value `i = 1, i = 2, ..., i = 6` using a *loop*. The simplest type of loop is called a `for` loop, which corresponds to the phrase "for the values of `i` in (some values), do (some expression)." The syntax of a loop is



```
for (i in values) {
  expressions
}
```

The layout of this is similar to when we wrote a function; we have the identifier `for` (similar to the function identifier), a statement in parentheses which instructs which values to use for `i` (similar to function arguments) and a set of expressions grouped together with curly braces (similar to the function *body*). When this is evaluated, R will assign the first value to `i` and evaluate `expression`, then assign the second value to `i` and evaluate the `expression` again, and so on until it has assigned all of the values to `i`, at which point it will end evaluation.



Notice that we don't have a return value when using a loop; the scoping rules which apply during loop operations are exactly the same as we had previously outside of functions. A loop simply instructs the program to repeat itself, it does not create a new environment in which temporary values will be created, so we have no environment from which we need to return, and no values to return on completion.

Everything that happens during the loop evaluation happens in the environment in which it was called. This includes the assignment of values to the variable being looped (in our example, `i`) — following the loop evaluation, that variable will retain its value.

```
for (i in 1:3) {
```

```

    ## do nothing
}
print(i) ## after the loop
#> [1] 3

```

For our earlier example comparing `test_nums`, we can *abstract* the index into a loop

```

for (i in 1:6) {
  print(test_nums[i] > 5)
}
#> [1] TRUE
#> [1] FALSE
#> [1] FALSE
#> [1] FALSE
#> [1] TRUE
#> [1] FALSE

```

Note that this doesn't create a vector, it merely prints the values to the `Console`. This is an important distinction. If instead we want to create a vector with these results, we can insert each result in (since we are looping over the index) provided that the object exists first

```

greater_than_5 <- c() ①
for (i in 1:6) {
  greater_than_5[i] <- test_nums[i] > 5
}

```

There are several ways to create the initial empty object, including

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

which creates an empty vector (containing no elements, so returns `NULL`); or

```
vector(length = 6)
#> [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

which creates a (by default, `logical`) vector of the required length; or

```
rep(NA, 6) ①
#> [1] NA NA NA NA NA NA
```

- ① `rep()` creates an object with its first argument repeated some number of times; specified by the second argument.

which creates a vector of `NA` values of the required length. Each of these will work in the above example as a vector into which values can be placed.



### On 'initial' objects

If you decide you really need to do so, be very careful about how you create an initial 'empty' vector into which you intend to insert values. If for some reason a value can't be inserted, what will the default value be? Will `NA` or `FALSE` make sense for your object? Will type-coercion be an issue?

Another all-too-common pattern is to create a small initial object and 'grow' it as needed, for example

```
some_values <- c()
for (i in 1:100) {
  some_values <- c(some_values, i) ①
}
```

- ① On each iteration, the `some_values` vector is copied into a vector one element larger.

In this case the vector `all_values` begins empty, and on each iteration it is copied into a slightly larger vector. This is computationally expensive, and even this simple example takes a good few hundred thousands times longer than simply creating the vector once and **updating** the value at each index, such as

```
all_values <- rep(NA, 100)
for (i in 1:100) {
  all_values[i] <- i
}
```

despite producing the exact same result

```
identical(all_values, some_values)
#> [1] TRUE
```

The looping variable can take any valid name, and it's still a good idea to name it carefully if it's meaningful. The values to be assigned to the variable we are looping over need not be such a simple sequence either; any vector (or list) can follow the `in` identifier. You may wish to loop over even values



```
for (evens in c(2, 4, 6, 8)) {
  expression
}
```

or certain letters, this time using a list of values to iterate over



```
for (vowel in list("a", "e", "i", "o", "u")) {
  expression
}
```

or generate the values as you go using the result of another expression

```
## 6 cylinder automatics in mtcars
for (auto_6cyl in row.names(mtcars[mtcars$cyl == 6 & mtcars$am == 1, ])) {
  print(auto_6cyl)
}
#> [1] "Mazda RX4"
#> [1] "Mazda RX4 Wag"
#> [1] "Ferrari Dino"
```

It hopefully goes without saying that this is a vastly inferior way (both in readability and evaluation speed) compared with the vectorised syntax. The few occasions on which this will be the structure you want to use include when one iteration depends on

another, or when you really need to remain in the global scope.

## 8.2 Wider and Narrower Loop Scope

You may wonder what the effect would be of changing the loop variable *inside* the loop. Of course, we can simply try it and find out.

If we have a loop which prints out the value of the loop variable

```
for (i in 1:3) {
  message("starting i = ", i)
}
# starting i = 1
# starting i = 2
# starting i = 3
```

and we update `i` inside the loop, the value of `i` at the start of each iteration is still the loop value

```
for (i in 1:3) {
  message("starting i = ", i)
  i <- 7
}
# starting i = 1
# starting i = 2
# starting i = 3
```

but takes the new value during the loop itself

```
for (i in 1:3) {
  i <- 7
  message("ending i = ", i)
}
# ending i = 7
#> ending i = 7
#> ending i = 7
```

When we write loops using a function (such as we do when using `purrr's map`) remember that any locally defined variables (within that function) will not persist outside of the looping.

They can however persist when we create a more narrow scope within a loop, such as another loop (or a function for that matter). If we (for whatever reason) wanted to loop over the rows and columns of a matrix, then in order to keep these iterators separate, we need different names for them. A logical choice is to name these based on that property. If we have a small matrix

```
m <- matrix(letters[1:6], nrow = 2, ncol = 3, byrow = TRUE)
m
#>      [,1] [,2] [,3]
#> [1,] "a"   "b"   "c"
#> [2,] "d"   "e"   "f"
```

and we wish to loop over both the rows and columns to print out some information, we

can nest our loops knowing that the innermost calls will have access to the widest scope, so all of the iterators will be valid

```
for (row in 1:2) {
  for (col in 1:3) {
    print(paste0("m[", row, ", ", col, "] = ", m[row, col]))
  }
}
#> [1] "m[1, 1] = a"
#> [1] "m[1, 2] = b"
#> [1] "m[1, 3] = c"
#> [1] "m[2, 1] = d"
#> [1] "m[2, 2] = e"
#> [1] "m[2, 3] = f"
```

Keep an eye out for examples that match these patterns and try to re-write them in a vectorised format.

### 8.2.1 `while` loops

Another type of loop is the `while` loop. Rather than specifying a list of values to loop over, a condition can be used; the loop will continue as long as the condition evaluates to TRUE. At the start of each loop iteration, the condition is evaluated and if it produces TRUE, the loop *body* is executed. When the *body* has finished executing, the next iteration begins with another evaluation of the condition.

This of course requires that the condition be updated through the execution of the body, otherwise the loop is infinite, and R will continue executing the loop until you explicitly cancel it (e.g. using the 'stop' button or pressing `Esc`)



```
while (237 > 1) {
  print("all work and no play makes Jack a dull boy")
}
#> [1] "all work and no play makes Jack a dull boy"
#> [1] "all work and no play makes Jack a dull boy"
#> [1] "all work and no play makes Jack a dull boy"
#> [1] "all work and no play makes Jack a dull boy"
#> [1] "all work and no play makes Jack a dull boy"
#> [1] "all work and no play makes Jack a dull boy"
#> [1] "all work and no play makes Jack a dull boy"
#> [1] "all work and no play makes Jack a dull boy"
#> [1] "all work and no play makes Jack a dull boy"
#> [1] "all work and no play makes Jack a dull boy"
#> <truncated>
```

If the condition *never* evaluates to TRUE then the body is never executed

```
while (1 == 2) {
  print("this line will not be printed")
}
```

The condition in the `while` statement needs to be valid, which means that any variables it depends on need to already exist. One way to deal with this is to be explicit about the first value by initialising it outside of the `while` statement. Within the body of the loop, the variable(s) can (and should) be updated. A simple `while` loop which prints the current value of `i` on each iteration then increases it by 1 until it is no longer less than 3 is

```
i <- 0      1
while (i < 3) { 2
  print(i)    3
  i <- i + 1  4
}
#> [1] 0
#> [1] 1
#> [1] 2
```

- 1 initialise `i` to 0
- 2 the condition of the loop is that `i` is less than 3, which it is to begin with
- 3 on each iteration, the value of `i` is printed
- 4 at the end of each iteration, `i` is raised by 1



### while loop scoping

As with the `for` loop construct, the operations are executed within the global scope, which means that any variables produced remain in the global scope once the loop is complete. In this case, the condition requires that the variables already exist since they need to be evaluated, but they remain as they were at the end of the loop execution

```
i <- 1      1
while (i < 4) { 2
  i <- i + 1  3
}
print(i)      4
#> [1] 4
```

- 1 prior to the `while` loop, `i` has the value 1
- 2 the condition here is that `i` is less than 4, so either 1, 2, or 3
- 3 within the loop body, `i` is increased by 1, even on the iteration where `i` starts with 3. Once this iteration is complete, the condition will be FALSE and the loop will be exited
- 4 after the loop, `i` has the value 4 because it had been increased by 1 on the final evaluation iteration.

This construct can be very useful when you are repeating some operation on external varying data (scraping a website which changes, or monitoring some external condition). Alternatively, you may wish to perform some operation(s) only when a specific condition is met.

## 8.3 Conditional Evaluation

Rather than looping over values, a condition can be tested on its own and some code executed if it evaluates to TRUE. This is extremely useful but also requires great care. There are several ways to specify conditional evaluation depending on the result you are trying to achieve.

### 8.3.1 `if` conditions

The most straightforward way to introduce some conditional evaluation into some code is to use the `if` construct. Here we again test a condition and if it evaluates to TRUE a statement will be executed

```
x <- 5
if (x > 3) {
  print("x is greater than 3")
}
#> [1] "x is greater than 3"
```

- ➊ Set the value of x to 5. This could just as easily be a value you don't necessarily know the value of yet.
- ➋ Test whether x is greater than 3; if it is, the statement between the curly braces will be evaluated.
- ➌ If the tested statement returns TRUE, this will be evaluated.

If the condition is FALSE then the statement is ignored entirely, even if it contained invalid code (thanks to R's "lazy evaluation" mechanism)

```
x <- 5
if (x < 2) {
  print(object_that_doesnt_exist)
}
```



### On testing missing logicals

The only values that the condition can validly evaluate to are TRUE and FALSE. If the condition evaluates to NA then an error will be raised

```
x <- NA
if (x > 3) {
  print("x is greater than 3")
}
```



Error: missing value where TRUE/FALSE needed

This gets messy when the condition depends on the result of further calculations which unexpectedly evaluate to NA, but those are scenarios for which you haven't planned and R doesn't know whether you want to evaluate the statement or not.

The condition doesn't need to be an expression of its own; it can simply be a value that is TRUE or FALSE

```
x <- TRUE
if (x) {
  print("do all the things")
}
#> [1] "do all the things"
```

An offshoot of this is when a value can be coerced to either TRUE or FALSE. Recall that this is true of many values, such as "TRUE", "T", and 1 evaluating to TRUE, and "FALSE", "F", and 0 evaluating to FALSE.

The braces are not *technically* required if your entire statement to be evaluated conditional on the test is a single statement

```
x <- 5
if (x > 3)
  print("x is greater than 3")
```

```
#> [1] "x is greater than 3"
```

but this is highly dangerous, because the restriction of "a single statement" can easily be forgotten and bugs introduced. If we have the following code in which a second line was added to the `if()` construct during some changes

```
x <- 5
if (x > 3) ①
  print("x is greater than 3")
  print("this should only appear when x > 3") ②

x <- 2
if (x > 3)
  print("x is greater than 3")
  print("this should only appear when x > 3") ③
```

- ① Neglecting to use the braces is valid if you have only a single statement following the `if()`, but this is dangerous.
- ② An earlier draft of this `if()` code had the single condition, but later a second one was introduced. It appears to still work fine when the condition is met.
- ③ When the condition is not met however, this line should *not* be evaluated, but it is because it is not actually part of the `if()` construct.

Evaluating the first part doesn't highlight any issues; the condition is met so both statements are printed

```
x <- 5
if (x > 3)
  print("x is greater than 3")
  print("this should only appear when x > 3")
#> [1] "x is greater than 3"
#> [1] "this should only appear when x > 3"
```

With the second construction though

```
x <- 2
if (x > 3)
  print("x is greater than 3")
  print("this should only appear when x > 3")
#> [1] "this should only appear when x > 3"
```

we see that the second message is printed (since it is not part of the `if()` construct at all), which was not the intention.

Sometimes we will wish to perform some other calculation when our condition does return FALSE. Rather than having to perform the test twice (for both `if (TRUE)` and `if (FALSE)`), an `else` statement can be added to this construct

```
x <- 5
if (x > 3) {
  print("x is greater than 3")
} else {
  print("x is less than 3")
}
```

```
#> [1] "x is greater than 3"
```

The `else` needs to immediately follow the closing curly brace of the `if` body for the construct to be valid. The curly braces themselves are still not required if the body in both cases consists of just a single line of commands/statement. The entire construct can actually be written on a single line if desired, though this quickly becomes unreadable if the code body is anything but simple

```
x <- 5
if (x > 3) print("x is greater than 3") else print("x is less than 3")
#> [1] "x is greater than 3"
```

We can even have more than one level of testing by adding *another* `if` statement to the `else`

```
x <- 5
if (x > 6) {
  print("x is greater than 6")
} else if (x > 4) {
  print("x is greater than 4")
} else {
  print("x is less than 5")
}
#> [1] "x is greater than 4"
```

Note that only the `"x is greater than 4"` result was printed; once a condition is tested and found to be TRUE, evaluation exits the `if()` construct entirely and the other conditions are not evaluated. This means that they don't even need to be valid, since they will be lazily evaluated

```
x <- 5
if (x > 3) { ❶
  print("x is greater than 3")
} else if (nonesense) { ❷
  print("how did I get here?")
}
#> [1] "x is greater than 3"
```

- ❶ This condition evaluates to TRUE so its statement will be evaluated.
- ❷ `nonesense` isn't a defined variable, but it is not evaluated in this case anyway. See what happens when it that statement *is* evaluated by changing the value of `x` and evaluating the block again.

These conditions are tested in the order they're written, and once one evaluates to TRUE then that condition's corresponding code will be executed and the testing concluded.



### On overlapping conditions

Even if some of the conditions have overlap, the testing will be completed in order and either zero or one of the code blocks will be executed (not multiple blocks). For example, if we tested whether `x = 3` is greater than 1 and greater than 2 (both being TRUE), then the order in which the tests are performed will still determine which code block is executed

```
x <- 3
if (x > 2) {
  print("x is greater than 2")
} else if (x > 1) {
  print("x is greater than 1")
} else {
  print("x is smaller than 1 and 2")
}
#> [1] "x is greater than 2"
```

**There is no limit to how many else statements are chained, but only zero or one of them will ever result in the corresponding code being executed.**

The code block which is potentially executed following an if condition can contain any code, be that assignment (to the global environment), function calls, or just a long series of calculations. An even more useful feature of the if construct is that it can itself be used to conditionally assign something to a variable

```
x <- 5
y <- if (x < 3) "small" else "big"
y
#> [1] "big"
```

which is equivalent to moving the assignment into the body of the statement, but which requires two separate assignments

```
x <- 5
if (x < 3) {
  y <- "small"
} else {
  y <- "big"
}
y
#> [1] "big"
```

### On multiple values in conditions

The if construct is designed to perform a single test; that is, the condition should return a single value of either TRUE or FALSE. We can however construct conditions which return multiple logical values, but these should not be used in the if construct condition. Doing so produces a warning

```
if (c(1, 2, 3) > 2) { print("TRUE") }
# Warning message:
# In if (c(1, 2, 3) > 2) {: the condition has length > 1 and only the
#> first element will be used
```

though the first element is still accepted and used as the result of the test. In future versions of R this will be an error since multiple values have no use in this construct.<sup>67</sup>

When we do wish to perform multiple conditional comparisons, there is a more

---

<sup>67</sup> You can enforce the error in recent versions of R by setting the environment variable `R_CHECK_LENGTH_I_CONDITION` to true.

suitable construct available.

### 8.3.2 *ifelse* conditions

An important feature of the `if` construct is that the condition *should* be of length one, but what if we wish to do something to each *element* of a vector depending on how it satisfies a condition? The `ifelse()` function is made for exactly this situation.

The arguments are structured similarly to the `if` and `else` identifiers; we specify a condition we wish to test (which can return multiple values), and replacements for when the result is TRUE or FALSE. This looks like



```
ifelse(test, yes, no)
```

where `test` is either a logical vector or a computation which produces one (or can be coerced to one); and `yes` (and `no`) are vectors of values to be used when `test` returns TRUE (or FALSE). These two outcomes need not be the same *type* or even length, but exactly what will be returned is highly dependent on the value of `test`. The value produced by `ifelse()` is just another value, and can be assigned to a variable.



**Great care needs to be taken with this function. The type of value returned depends on whether the test results in yes or no being evaluated. We can call this function with two very different return options**

```
ifelse(TRUE, "a string", 7L)
#> [1] "a string"
ifelse(FALSE, "a string", 7L)
#> [1] 7
```

There is of course a third option; the test can result in NA in which case the result will also be NA

```
ifelse(NA, "test was TRUE", "test was FALSE")
#> [1] NA
```

This can lead to issues if the value of `test` may change and thus alter the type of the return value.

Due to 'lazy evaluation', if the test never results in one or both of yes or no then R will never attempt to evaluate those arguments. This means we don't have to worry about undefined values if they are not going to be needed

```
ifelse(FALSE, undefined_variable, 8L)
#> [1] 8
```

The `test` argument of `ifelse()` can be a vector of length greater than one (compare this to the `test` in an `if()` construct which *should* be of length 1, and produces a warning otherwise), and a consequence of this is that the result will be of that length. This makes sense; each element is tested and either `yes` or `no` arguments or `NA` will be evaluated for each result. The arguments `yes` and `no` however can be of any length (either shorter or longer than `test`) but only the corresponding values will be used (or

the result will be *recycled*)

```
ifelse(TRUE, c(2, 3), "z") ①
#> [1] 2
```

- ① Here the yes argument is longer than test, so only the first value is used.

```
ifelse(c(TRUE, TRUE), 3, "z") ①
#> [1] 3 3
```

- ① Here the test argument is longer than yes so yes is *recycled*.



If any element of either yes or no *will* be needed, then the entire vector will need to be valid (i.e. contain existing variables). When test is a vector, the corresponding elements from yes and no will be extracted, but those vectors will need to be evaluated before that happens. For example, if we were going to extract the first value of yes and the second of no (both 3 in this example)

```
ifelse(c(TRUE, FALSE), c(3, 9), c(9, 3)) ①
#> [1] 3 3
```

- ① Here the test is TRUE for the first element, so returns the first element of yes. It is FALSE for the second element, so returns the second element of no.

When the yes vector cannot be constructed however (because a variable is undefined) then it fails

```
ifelse(c(TRUE, FALSE), c(3, not_a_var), c(9, 3))
```



Error: object 'not\_a\_var' not found

This becomes particularly important when we wish to perform an operation on a vector conditionally based on the specific values. If we have a vector which has both positive and negative values

```
x <- -3:3
x
#> [1] -3 -2 -1  0  1  2  3
```

we can try to take the square root of this

```
sqrt(x)
# Warning message:
# In sqrt(x): NaNs produced
#> [1]      NaN      NaN      NaN 0.000000 1.000000 1.414214 1.732051
```

The warning is because we can't really take the square root of a negative number, so the result contains NaN ('not a number'). This seems like a good candidate for `ifelse()`. We only want to take the square root of x when the value is non-negative. One option would be to use `x >= 0` (`x` greater than or equal to 0) as the test and `sqrt(x)` as the yes argument, since that's the result we want when the test is TRUE. We could use NA as the no argument so we don't get NaN there. When we try that, we get the result as we expected (`sqrt(x)` for non-zero x and NA otherwise) but we still generate the warning about NaN

```
ifelse(x >= 0, sqrt(x), NA)
```

```
# Warning message:
# In sqrt(x): NaNs produced
#> [1]      NA      NA      NA 0.000000 1.000000 1.414214 1.732051
```

The reason is that `sqrt(x)` is still evaluated for all elements; we're just keeping the ones that make sense in our result. A better way to achieve this is to *only* pass sensible values to `sqrt()` by performing the `ifelse()` step first

```
sqrt(ifelse(x >= 0, x, NA))
#> [1]      NA      NA      NA 0.000000 1.000000 1.414214 1.732051
```

since `sqrt(NA)` doesn't produce a warning (it just returns NA).

We can combine this `ifelse()` construct with `dplyr` operations since it is a vectorised operation. If we want to create a new column in a `data.frame` where the value depends conditionally on some other column, we can do so using `mutate()` and `ifelse()`

```
head(
  dplyr::mutate(
    mtcars,
    hp_per_cc_auto = ifelse(am, hp / disp, NA)
  )
)
#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb hp_per_cc_auto
#> 1 21.0   6 160 110 3.90 2.620 16.46  0  1    4     4    0.6875000
#> 2 21.0   6 160 110 3.90 2.875 17.02  0  1    4     4    0.6875000
#> 3 22.8   4 108  93 3.85 2.320 18.61  1  1    4     1    0.8611111
#> 4 21.4   6 258 110 3.08 3.215 19.44  1  0    3     1           NA
#> 5 18.7   8 360 175 3.15 3.440 17.02  0  0    3     2           NA
#> 6 18.1   6 225 105 2.76 3.460 20.22  1  0    3     1           NA
```

- ➊ Only print() the first six rows.
- ➋ 'Modify' the `mtcars` data.frame using `dplyr`.
- ➌ Create a new column `hp_per_cc_auto`, referencing columns from `mtcars` using Non-Standard Evaluation.
- ➍ The `ifelse()` uses `am` (does the vehicle have automatic transmission?) as the test where 1 indicates TRUE and is coercible as such).
- ➎ Where the test value is TRUE, divide the `hp` value by the `disp` value, otherwise return NA.

Passing other objects (e.g. list, matrix) is possible but take note that the condition that the result be the same size and shape as `test` will remain.

There is a handy `dplyr` way to achieve this task; the `case_when()` function allows us to use multiple conditions to produce different results, and works inside a `mutate()` call

```
head(
  dplyr::mutate(
    mtcars,
    hp_per_cc_auto = dplyr::case_when(am == 1 ~ hp / disp, ➊ ➋
                                         am == 0 ~ NA_real_) ➌ ➍
  )
)
#>   mpg cyl disp  hp drat    wt  qsec vs am gear carb hp_per_cc_auto
```

```
#> 1 21.0   6 160 110 3.90 2.620 16.46 0 1 4 4 0.6875000
#> 2 21.0   6 160 110 3.90 2.875 17.02 0 1 4 4 0.6875000
#> 3 22.8   4 108 93 3.85 2.320 18.61 1 1 4 1 0.8611111
#> 4 21.4   6 258 110 3.08 3.215 19.44 1 0 3 1 NA
#> 5 18.7   8 360 175 3.15 3.440 17.02 0 0 3 2 NA
#> 6 18.1   6 225 105 2.76 3.460 20.22 1 0 3 1 NA
```

- ➊ when using `case_when()` inside `mutate()` we can use column names without prefixing them with the name of their data.frame.
- ➋ The syntax for conditions is a comma-separated set of tests and results written as formulae; `<test> ~ <result>`.
- ➌ All results must explicitly be of the same type, so we can't just use NA, we need the explicit `NA_real_`.
- ➍ To allow for a 'catch-all' condition, the last condition can be `TRUE ~ <result>`.

## 8.4 Try It Yourself

 Let's work through an example together. Let's say we have the unique values of `cyl` from the `mtcars` object

```
## find the unique values of cyl in mtcars and sort them numerically
## (by default they are in the order in which they first appear)
car_cyls <- sort(unique(mtcars$cyl))
```

and we wanted to produce some summary output of the most efficient (in terms of miles per gallon) vehicle in each cylinder class. We could of course produce an object of these (using `dplyr`) but let's make a function that writes a summary to the screen. Read through the following function to make sure you understand what it's doing

```
## takes input num_cyls and prints a statement
## about mtcars rows with that number of cylinders
showcase <- function(num_cyls) {

  ## subset mtcars to the rows with num_cyls cylinders
  ## this uses [ instead of dplyr::filter since the latter
  ## drops row names
  cars_with_cyls <- mtcars[mtcars$cyl == num_cyls, ]

  ## identify the row with the highest mpg
  ## uses [ to preserve rownames
  ## which.max finds the index of the highest value
  most_efficient <- cars_with_cyls[which.max(cars_with_cyls$mpg), ]

  ## print a nice message to the console explaining the results
  cat(paste("There are", nrow(cars_with_cyls), "cars with", num_cyls,
            "cylinders.\nThe most fuel efficient of these is the",
            rownames(most_efficient), "\n\n"))

}
```

We can test this function with an example

```
showcase(4)
#> There are 11 cars with 4 cylinders.
```

```
#> The most fuel efficient of these is the Toyota Corolla
```

Now, if we wanted to produce this output for each of the classes in `car_cyls` we might think to loop over those values. Instead, we can have `purrrr` iterate over them for us

```
# install.packages("purrr")
library(purrr)
## walk is equivalent to map but doesn't return anything explicitly
## (actually returns the input list, invisibly)
walk(car_cyls, showcase)
#> There are 11 cars with 4 cylinders.
#> The most fuel efficient of these is the Toyota Corolla
#>
#> There are 7 cars with 6 cylinders.
#> The most fuel efficient of these is the Hornet 4 Drive
#>
#> There are 14 cars with 8 cylinders.
#> The most fuel efficient of these is the Pontiac Firebird
```

## 8.5 Summary

In this chapter we've seen several ways in which we can iterate over values. Hopefully it's clear that the `purrrr`-framework of mapping over a function is superior. Don't worry if you find yourself trying to write explicit loops; but when you do, try to re-frame what you have in terms of a function and a map.

You've learned that:

- Vectorisation is a superior way to loop through elements of a structure
- The `purrr` package's `map()` function (and associates) makes repetition clean, safe, and fast
- Anonymous functions can save having to explicitly create a function
- Variables created during explicit loops remain in the current scope
- Conditional evaluation can be achieved using `if()` and `ifelse()`

New terms you've learned:

### **loop**

a repetition of some code, often replacing a particular value each time around.

### **abstraction**

replacing some specific value with a more general placeholder.

### **side effect**

something that happens not directly related to the data involved, such as producing a plot, interacting with an external system, or printing something to the Console.

### **type safety**

the notion that the return value(s) of a function will always be of an expected *type*. The lack of type safety of some functions makes working programatically with data complex.

Things to remember:

- Loops can often be replaced with faster and simpler vectorization.
- Temporary (anonymous) functions can be created inside map using a formula; `~ .x + 1` is shorthand for `function(x) x + 1`.
- Variables created within loops are still in the global namespace after evaluation of the loop is complete.
- Conditions used in calls to `if()` should only be a single element (length 1).
- The result of a call to `ifelse()` will be the same size and shape as the condition tested, with values taken from the yes or no arguments.

With the tools you now have, you should be able to work with your raw data to produce some new version of it which helps you to answer your original questions. In order to elucidate those answers however, we often need to visualise the relationships between different parts of data. To accomplish that, we need to generate some *plots*.

# I Want I Want To Visualise My Data

## (Plotting)

**This chapter covers:**

- Preparing data for visualisations
- Plotting data the tidy way; `ggplot2`
- Saving graphics to files

Visualisations of your data help both you and people with whom you share your results gain insight into relationships inherent in the data. When we have many data points (even just a handful) a *plot*(graph, graphic, or some other sort of image) can convey a great deal of information which a table simply can't. This isn't to say that a plot is *always* the best way to convey information, but when it is, we need some tools to generate these easily and effectively with **R**.

## 9.1 Data Preparation

By now you've taken your *raw* data, performed some operations on it (cleaned, sliced, and diced), and are ready to plot it to see what secrets it might be able to tell you. The optimum arrangement of your data in terms of joining together pieces of information may not however be the optimum arrangement in terms of thinking about re-slicing it for visualisation.

It's not uncommon to need to generate another version of a dataset specifically for visualisation, but that should be reproducibly derivable from both the original *raw* data and any subsequent transformations performed.

When we have various groups in data, one useful way to include this information in a visualisation is to visually separate the groups, either by colour, by shape, by linetype, or by physical distance. This means that our data needs to be in a format which is amenable to such splitting.

### 9.1.1 Tidy Data, Revisited

We've discussed 'tidy data' a few times now and visualisations will lead us to yet another justification for maintaining these principles in our data. The idea of 'wide' data with 'one row per subject, one column per observation' means that grouping together observations with something in common (e.g. all the automatic transmission vehicles) becomes as simple as filtering values of just one column.

If our data was in the 'long' data format (few columns, and a variable describing to which observation a measurement refers) then this filtering becomes much more complex. If we wish to group observations, we need to filter both the column that describes variables and the column of measurements. This quickly becomes unwieldy. Tidy data is therefore a much more suitable format for visualisation.

### 9.1.2 Importance of Data Types

We covered the different data types early on, and perhaps now you've seen examples where the specific *type* in which a column of data is stored alters the analysis performed on it. This becomes particularly important when filtering values, which is an integral part of subsetting and grouping.

Many of the visualisation tools we are about to see are as easy to use as they are because they automate a lot of common tasks for us. When a data type is *continuous* (can take any value, not just some *discrete* categories) the plots will reflect this quality. When we do need categories however, the plots produced will be constructed differently, in a way that reflects this quality instead. We won't (necessarily) need to tell R to use a different tool for these different types; the tools have been *abstracted* to allow either, and do something different depending on which data type is present. This can of course be overridden, which we'll also cover.

There is then a high importance to place on ensuring that our data is in the correct format before we attempt to visualise it. Does the quantity we wish to use as the *grouping variable* (the quantity which defines which group a measurement belongs to, in order to somehow distinguish these in a plot) relate to a continuous value (say, numeric) or a discrete value (a factor)? These two options produce visually different plots.

## 9.2 *ggplot2*

There are many different visualisation tools available for R, but one of the most popular and sophisticated is *ggplot2*. This is the second incarnation of the 'grammar of graphics' plotting system developed by Hadley Wickham as an implementation of Leland Wilkinson's Grammar of Graphics.<sup>68</sup>

The general concept that this package encapsulates is that of a 'grammar'; an explicit language which connects your *data* (the first and foremost feature in a visualisation) to what you see on the screen as *aesthetic mappings*. In a spreadsheet you would perhaps

<sup>68</sup> Wilkinson, Leland (2005). The Grammar of Graphics. Springer. ISBN 978-0-387-98774-3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

select the values you wish to have on the x and y axes, produce a plot, then adjust colours and settings to achieve the plot you really want. In `ggplot2`, these settings are directly connected to the data through the *mappings*; e.g. when plotting columns from `mtcars`, determining the colour of points by whether or not the vehicle was automatic transmission, as defined by corresponding values in the column `am`.

The other primary features of `ggplot2` are the types of features we wish to plot (formally *geometries*) and the relationships between values within a group or aspect (formally *scales*).

These explicit mappings mean we have more control over how plots are generated, but of course with such sophistication we have a syntax to become accustomed to.

### 9.2.1 General Construction

The four features we mentioned above; data, geometries, aesthetic mappings, and scales; need to be presented to the `ggplot2` code in a particular way. For better or worse, `ggplot2` takes full advantage of the non-standard evaluation framework we saw in `dplyr`, which means that we can refer to columns of a `data.frame` or `tibble` in isolation within a `ggplot2` call.

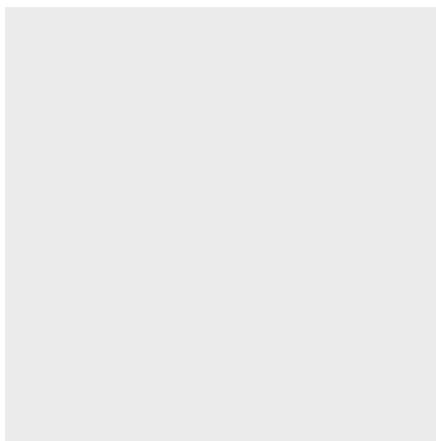
We begin of course by installing, loading and attaching the `ggplot2` package

```
# install.packages("ggplot2")
library(ggplot2)
```

The first ingredient to constructing a `ggplot2` visualisation is to request one, with the `ggplot()` function. Doing so with no further information produces an empty plot in the Plot tab, as in Figure 9. 1.

```
ggplot()
```

**Figure 9.1. An empty ggplot() call. No data was provided so nothing to plot.**



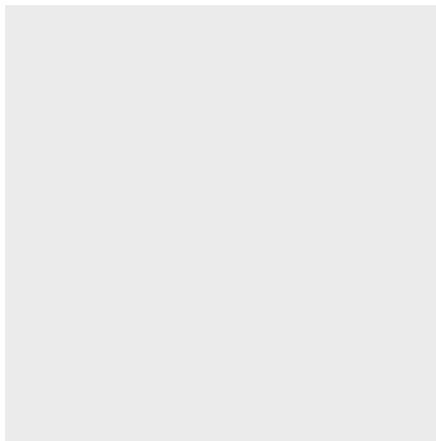
Note however that the plot area is not simply empty; a grey background is present.

Depending on how you have your panes set up in RStudio, the width and height of this may be different to what is shown here.

This is the default for `ggplot()`. Let's add some data (Figure 9. 2.).

```
ggplot(data = mtcars) ①
```

**Figure 9.2. Adding some data but still nothing to see here yet.**



① Once we're more familiar with the arguments, we will stop explicitly naming them.

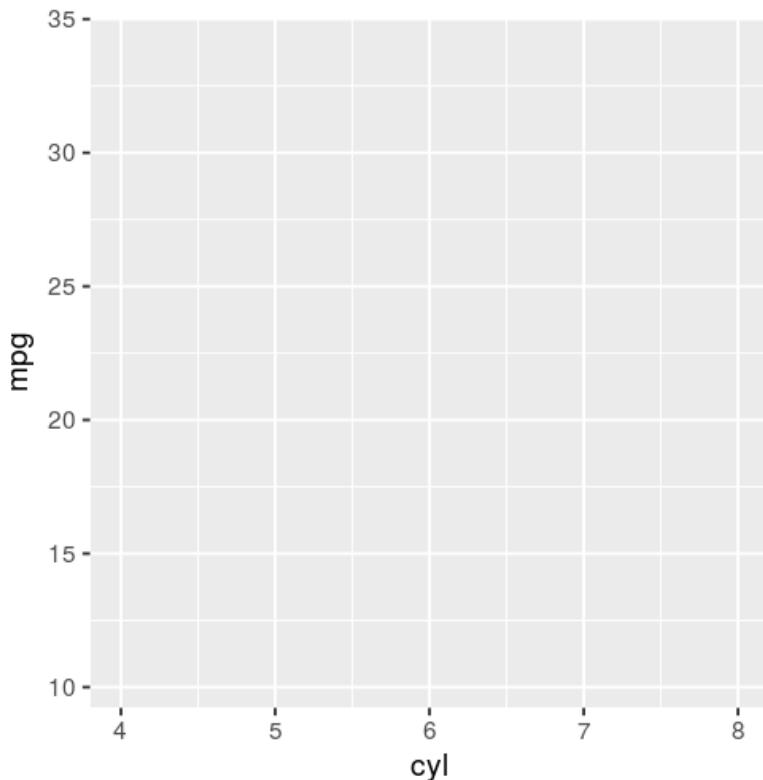
Nothing appears to have changed. `ggplot()` knows about our data, but not what we intend to *map* from data to the plot. To do that, we specify the mapping using the `aes()` function (for aesthetic). This takes bare column names and generates the relevant mapping for us.<sup>69</sup> If we intend to plot the mileage (`mpg`) on the y axis, and the number of cylinders (`cyl`) on the x axis, we can provide these using `aes()` as the `mapping` argument (Figure 9. 3.).

```
ggplot(data = mtcars, mapping = aes(x = cyl, y = mpg))
```

---

<sup>69</sup> `aes()` also performs some validation on the values provided such as autocompleting and autocorrecting potential aesthetic argument names.

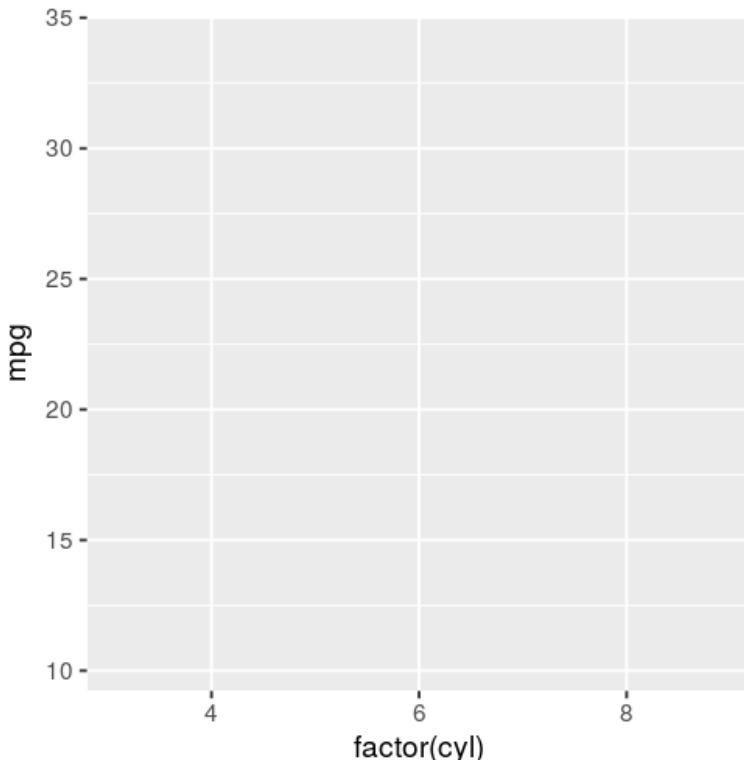
**Figure 9.3. Adding a mapping using the aesthetic function `aes()`. The axes are now well defined but we haven't specified what we want plotted.**



Now something is different; the axes have been titled with the relevant variable name and the axes themselves have been arranged and labelled according to the values in `cyl` and `mpg`. Note that since `cyl` was provided as it appears in the `mtcars` object (a numeric vector which only has values 4, 6, and 8) the default presentation of the x axis is to allow for any value in this range, with some automation of how to select the major and minor grid lines. If, instead, we wished to treat this as a category (which it really is) then we can modify the aesthetic in place (Figure 9.4.).

```
ggplot(data = mtcars, mapping = aes(x = factor(cyl), y = mpg))
```

**Figure 9.4. Using `factor()`` in the aesthetic we can change how the axis is constructed.**



Now the grid lines align with the only three possible values. Note also that the title for the x axis reflects what we have provided within `aes()`. We will see how to override this later.

We still don't have any paired values on the plot. This is because we haven't specified the *geometry* we want. Each geometry begins with the prefix `geom_`; if we wish to plot points, we can do so with `geom_point()` (Figure 9.5.).

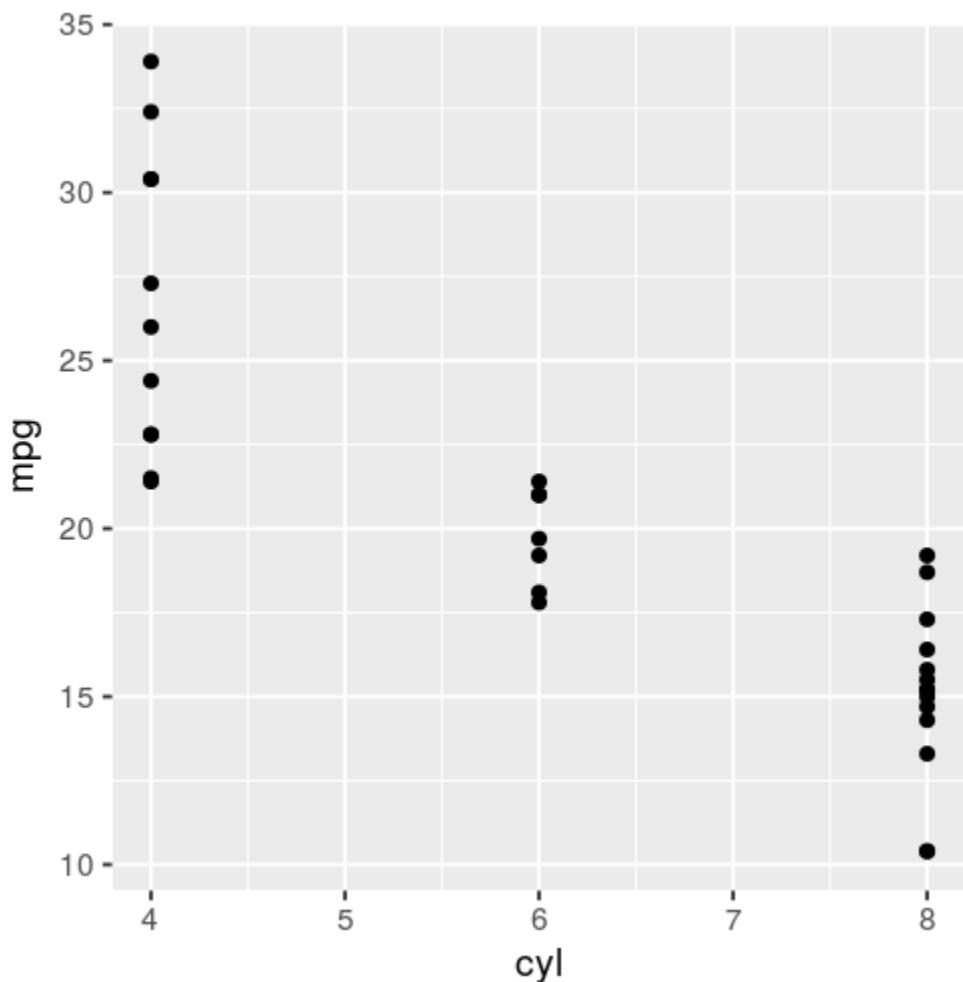


One thing to keep in mind is that ggplot2 has a 'pen and paper' model of constructing a plot; we begin with an empty canvas, and add subsequent elements to it one or more at a time. These are added using `+` which ggplot2 extends such that it behaves appropriately when it encounters ggplot2's class system (objects of class `gg`, `ggplot`, and related).

Plots themselves are stored just as anything else; as objects. This means that they can be saved (e.g. with `saveRDS()`), recalled, and modified. One limitation due to the 'pen and paper' model however is that elements cannot be removed once the object is generated. We can certainly remove their addition from the code, but we cannot remove them from a generated object.

```
ggplot(data = mtcars, mapping = aes(x = cyl, y = mpg)) + geom_point()
```

**Figure 9.5. Adding points as a geometry; with data a mapping and a geometry we finally have some data plotted.**



Once you're familiar with the syntax, it's much cleaner to drop any unnecessary variable names and wrap calls in a pipe-like manner, which produces the same result as above.

```
ggplot(mtcars, aes(cyl, mpg)) +
  geom_point()
```

The general construction of a `ggplot()` call typically follows the pattern shown in Figure 9. 6.

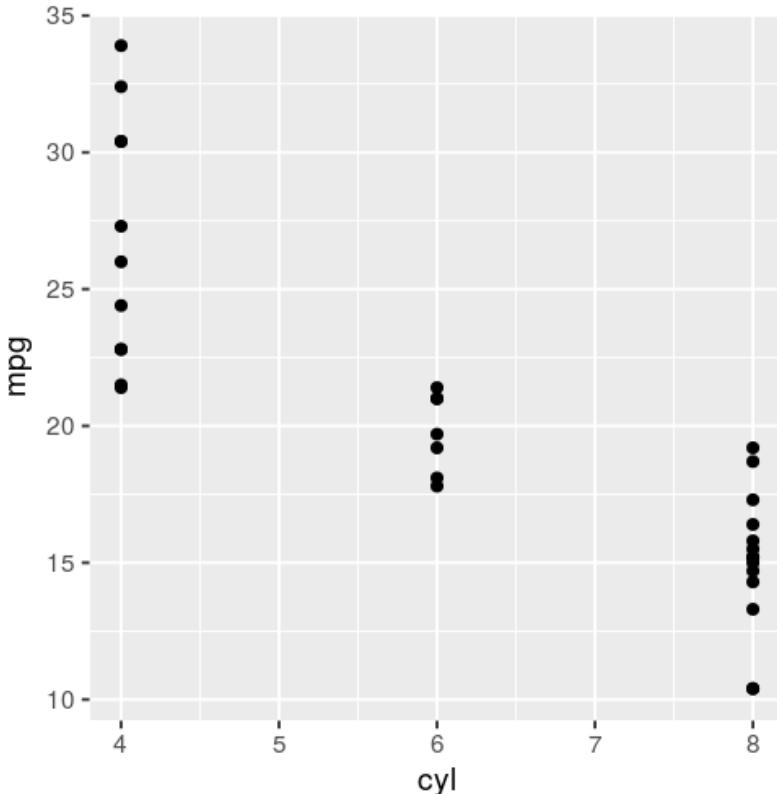
**Figure 9.6. ggplot2 call construction**

```
ggplot(data, aes(x, y)) + geom_X()
    ^ data
    ^ aesthetics
    ^ geometry
```

## 9.2.2 Adding Points

We've already added some points to our plot with a simple `geom_point()` call. This has taken advantage of the fact that our `ggplot()` call included data and the aesthetics we wanted `geom_point()` to use. Alternatively, we can specify these within the `geom_point()` call itself if we wish (Figure 9.7.).

```
ggplot() + geom_point(data = mtcars, aes(cyl, mpg))
```

**Figure 9.7. Using aes() in geom\_point()**

Note that I've specified the `data` argument explicitly; this is because the arguments

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

Licensed to Asif Qamar <[asif@asifqamar.com](mailto:asif@asifqamar.com)>

for `geom_point()` are, in order, `mapping` *then* `data`. This means that failing to provide any argument names implies that we wish to use the data as the aesthetic. Thankfully, the error message suitably reminds us of this fact.

```
ggplot() + geom_point(mtcars, aes(cyl, mpg))
```

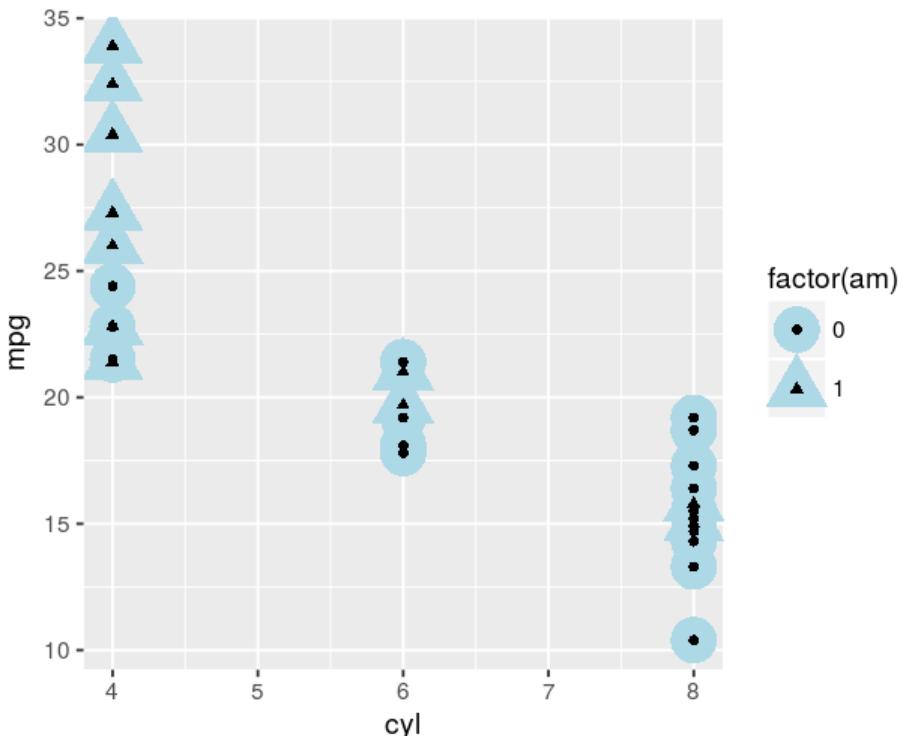


Error: ggplot2 doesn't know how to deal with data of class uneval

Using the `data` and `aes` arguments in the `ggplot()` call itself has an advantage; the subsequent `geom_` calls inherit these values, so we can add several with the same arguments passed on, but any new arguments being only local to that call (figure 9.8.).

```
ggplot(mtcars, aes(cyl, mpg, shape = factor(am))) + ①  
  geom_point(size = 8, col = "lightblue") + ② ③  
  geom_point() ④
```

**Figure 9.8. Aesthetics can be inherited by geometries if they are provided in the `ggplot()` call.**

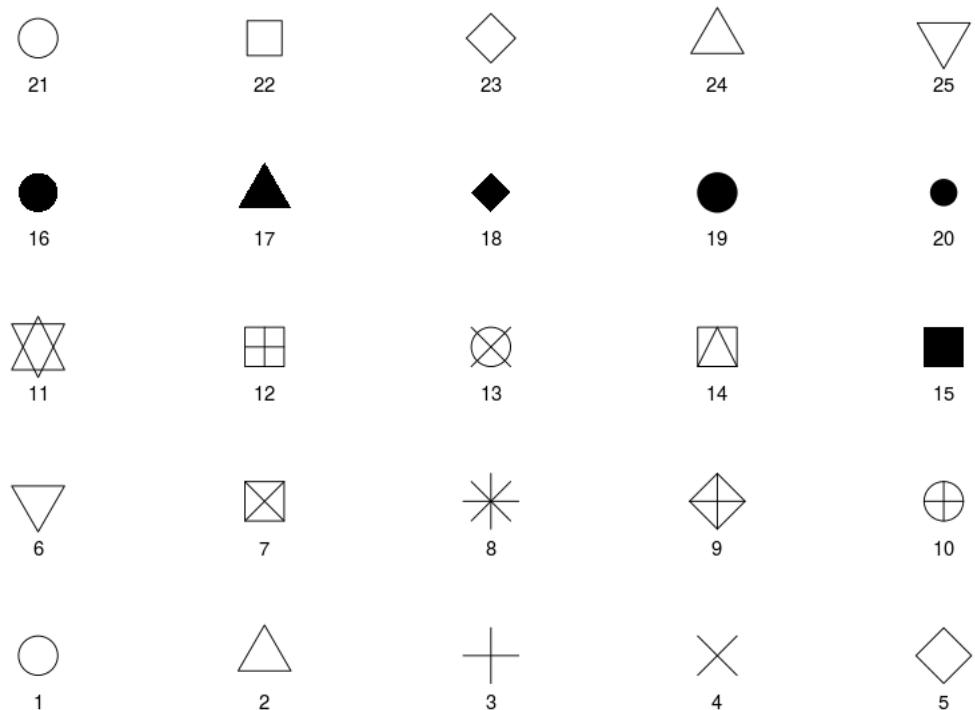


- ① The shape aesthetic will be inherited by all `geom_`s unless overridden.
- ② This call to `geom_point()` produces large, light blue points.
- ③ Note that we don't specify the shape in the `geom_` calls, these are inherited.
- ④ This call to `geom_point()` produces the default points, on top of the earlier points.

Because `ggplot()` follows the 'pen and paper' model of plotting, the order in which we request the `geom_s` determines the order in which they are drawn. If you have features you wish to appear above or below other features then change the order in which they are added appropriately.

The point shapes which can be plotted are numbered 1 to 25 and are shown in Figure 9.9. The default point shape is 16 which produces a small filled circle.

**Figure 9.9. Plotting points with pch values. The default point shape is number 16.**



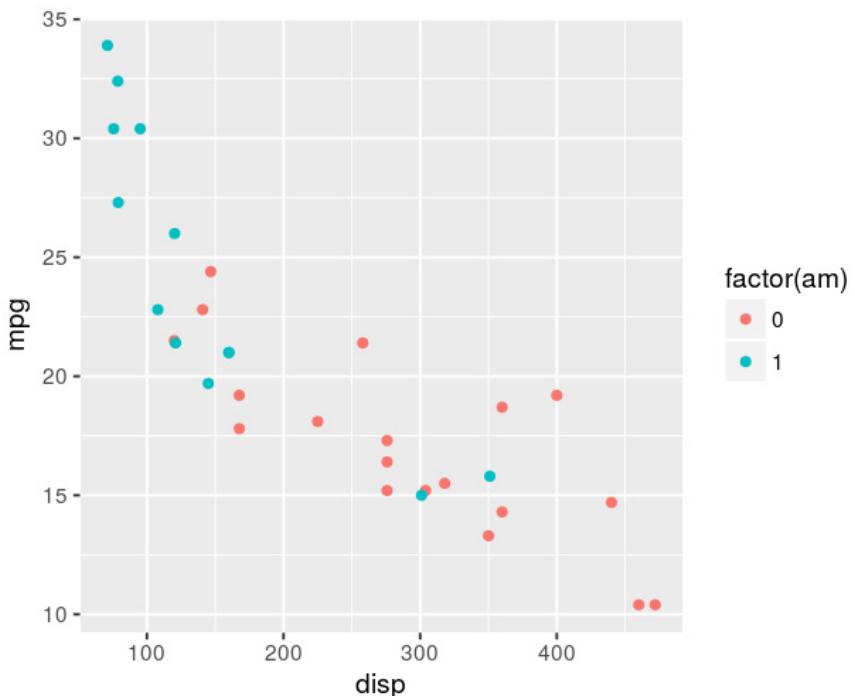
### 9.2.3 Style Aesthetics

The point symbol `shape` in the previous example is an example of styling. The styling can be added either to individual `geom_` calls for all members of that `geom_` (such as the `size` and `col` styling above); to individual `geom_` calls grouped by some quantity (within `aes()`, such as the `shape` styling above); or to all compatible `geom_` calls as an aesthetic (within the `ggplot()` `aes()` call, to be inherited by the `geom_s`). This allows a great deal of flexibility in how we construct illustrative plots.

If we want to make clearer the distinction between manual and automatic transmission vehicles when visualising the `mpg` at values of `disp` in the `mtcars` dataset, we can add some colour (distinguished by the value of `am`; the transmission column); Figure 9.10.

```
ggplot(mtcars, aes(disp, mpg, col = factor(am))) + geom_point()
```

**Figure 9.10. Using colour as an aesthetic via aes().**



Again I've used `factor(am)` to let `ggplot()` know that we wish to treat these as categories, despite them currently being numbers.

We can have many colours in the grouping, and `ggplot()` will automatically select these for us from a suitable set of pre-defined colours.<sup>70</sup> If we have three groups, the colours are slightly different to before, in Figure 9.11.

```
ggplot(mtcars, aes(disp, mpg, col = factor(cyl))) + geom_point()
```

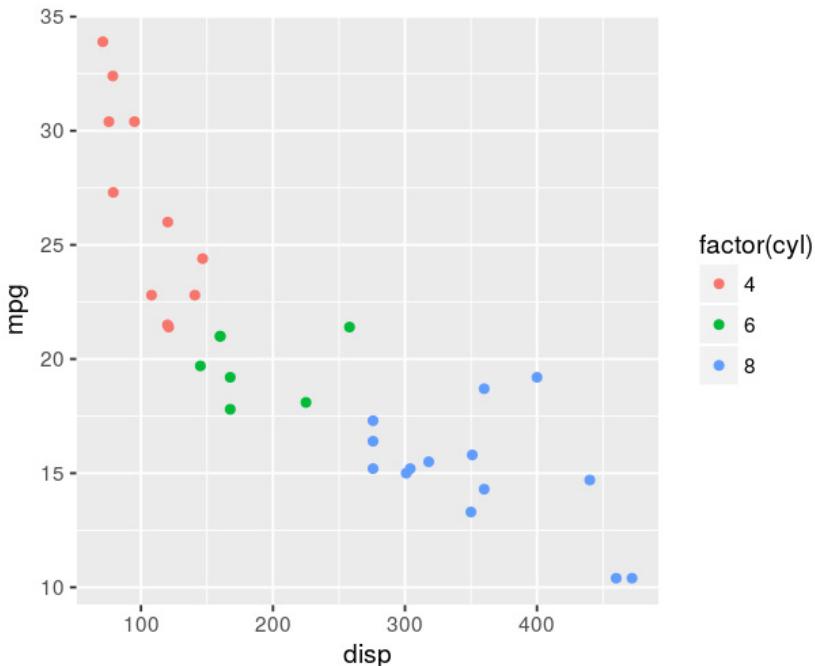
---

<sup>70</sup> In the current version of `ggplot2` these are chosen to be equally spaced hues around a colour wheel, starting at 15 degrees.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

**Figure 9.11.** Note the cyl vs disp relationship that becomes apparent.

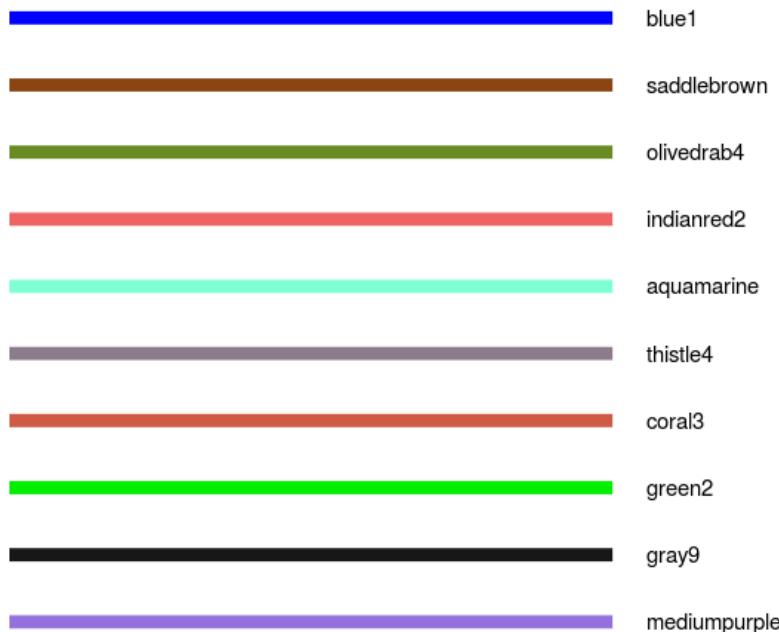


Choosing a colour ourselves to specifically use can be quite tricky, as there are many named colours from which we can select. The full list of named colours can be found using `colors()` (or `colours()`), a mere sample of which is

```
set.seed(13) ①
sample(colors(), 10) ②
#> [1] "mediumpurple" "gray9"      "green2"      "coral3"
#> [5] "thistle4"     "aquamarine"   "indianred2"  "olivedrab4"
#> [9] "saddlebrown"   "blue1"
```

- ① This seed was chosen because the random selection shows a wide variety of the available colours.
- ② `sample()` takes a random selection from a vector without replacement. If you want random selection with replacement then set `replace = TRUE`.

which are shown in Figure 9.12.

**Figure 9.12. Examples of colours available to ggplot().**

Any colour at all can be used if you specify either its RGB combination using `rgb()`,<sup>71</sup> or many of the other specifications for colours. Reading through `?colors()` is a good place to start.

The complete list of styles/aesthetics that can be applied is available with a non-exported (not-intended for use within your code) function

```
ggplot2:::all_aesthetics
#> [1] "adj"           "alpha"          "angle"          "bg"            "cex"
#> [6] "col"           "color"          "colour"         "fg"            "fill"
#> [11] "group"        "hjust"          "label"          "linetype"       "lower"
#> [16] "lty"           "lwd"            "max"            "middle"         "min"
#> [21] "pch"           "radius"         "sample"         "shape"          "size"
#> [26] "srt"           "upper"          "vjust"          "weight"         "width"
#> [31] "x"              "xend"          "xmax"          "xmin"          "xintercept"
#> [36] "y"              "yend"          "ymax"          "ymin"          "yintercept"
#> [41] "z"
```

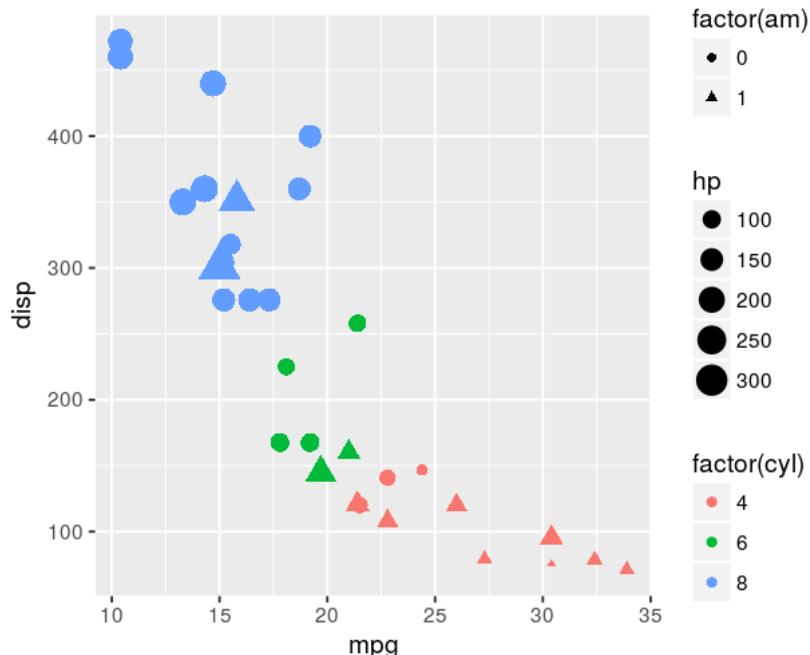
and many of these are straightforward enough to guess at their usage. Not every aesthetic can be applied to every `geom_` but the most commonly used ones (changing colour, linetype, size, fill colour, or transparency (`alpha`)) are generally available.

We can add as many of these as we wish to help convey the information in our data as another dimension to the visualisation (Figure 9.13.).

<sup>71</sup> Red, Green, Blue, in the range 0–255.

```
ggplot(data = mtcars,
       mapping = aes(x = mpg,
                      y = disp,
                      col = factor(cyl),
                      shape = factor(am),
                      size = hp)
) +
  geom_point()
```

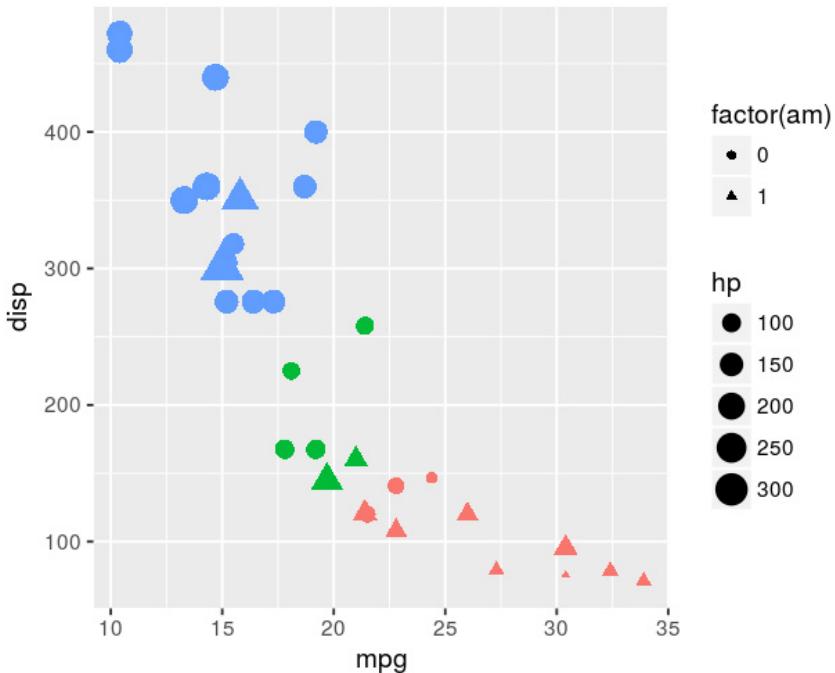
**Figure 9.13. More dimensions (as aesthetics) help to distinguish different aspects of the data and highlight relationships.**



When groupings are added via `aes()`, a 'legend' is produced alongside the plot to guide us as to which symbol/shape/colour represents which group. These too can be adjusted, for example removing the colour legend, using `guides()` (Figure 9.14.).

```
ggplot(data = mtcars,
       mapping = aes(x = mpg,
                      y = disp,
                      col = factor(cyl),
                      shape = factor(am),
                      size = hp)
) +
  geom_point() +
  guides(colour = FALSE) # no colour Legend
```

**Figure 9.14.** Removing a legend using `guides()`.



## 9.2.4 Adding Lines

If we wish to add one or more lines to the plot as a geometry, we can do so using `geom_line()`. For this example we need some data more easily represented by a line than the `mtcars` values, so we can create some

```
# install.packages("dplyr")
library(dplyr) ①
#
#> Attaching package: 'dplyr'
# The following object is masked from 'package:switchr':
#>
#>     location
# The following objects are masked from 'package:stats':
#>
#>     filter, lag
# The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

sin_df <- data.frame(theta = seq(pi, 0, -0.5)) %>% ②
  mutate(x = cos(theta), y = sin(theta)) ③
```

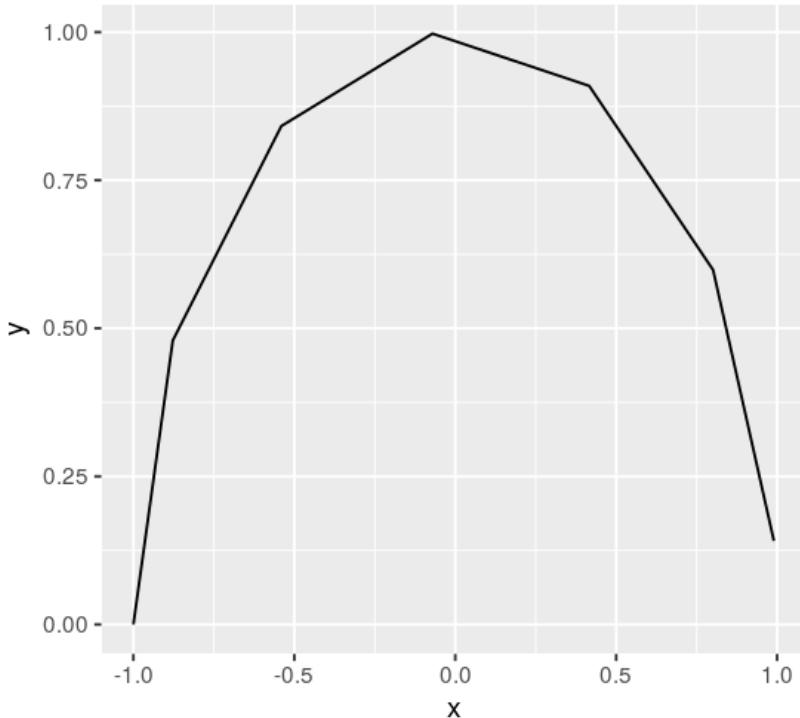
- ① We can use what we have learned about `dplyr` to make this easier.
- ② We start with a simple `data.frame` containing values of an angle, `theta`.

- ② Taking advantage of `dplyr`'s ability to re-use data, we can create two new columns which are derived from `theta`. The `x` and `y` values represent points on a circle.

This can be plotted as just lines joining the pairs of `x` and `y` values, as in Figure 9. 15.

```
ggplot(sin_df, aes(x, y)) + geom_line()
```

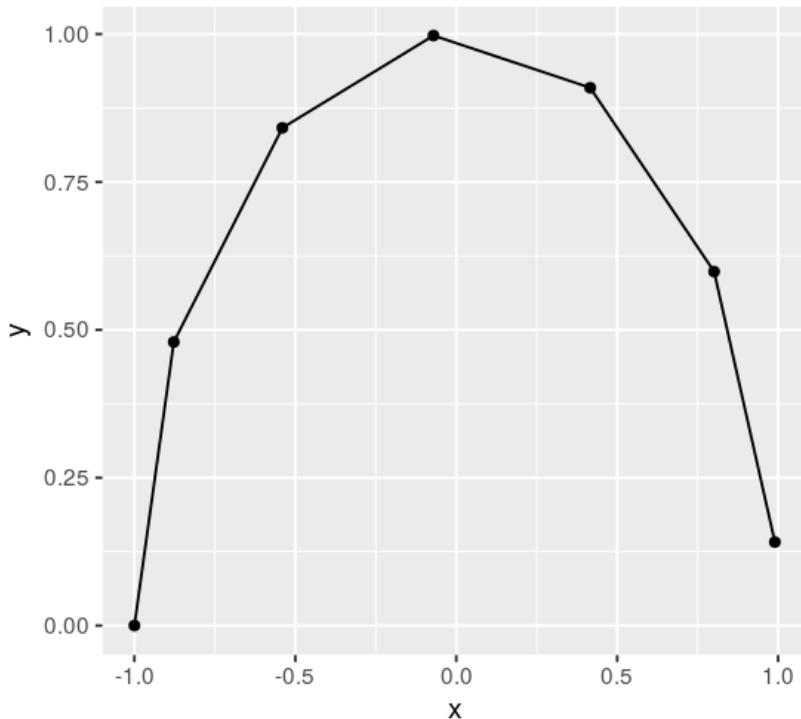
**Figure 9.15. Lines joining pairs of x and y values.**



Or we can specifically add the points in also (Figure 9. 16.)

```
ggplot(sin_df, aes(x, y)) + geom_line() + geom_point()
```

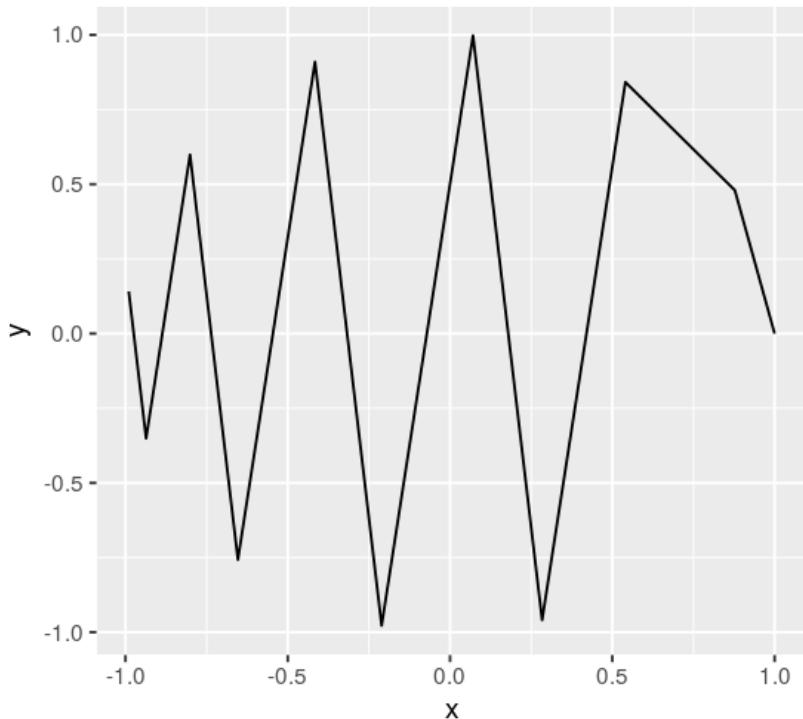
**Figure 9.16. Lines and points joining pairs of values.**



If we construct our data to cover more of a circle, we run into the problem that `ggplot()` tries to join up the paired values in a way we perhaps don't want (Figure 9.17.)

```
sin_df <- data.frame(theta = seq(0, 1.75*pi, 0.5)) %>%
  mutate(x = cos(theta), y = sin(theta))
ggplot(sin_df, aes(x, y)) + geom_line()
```

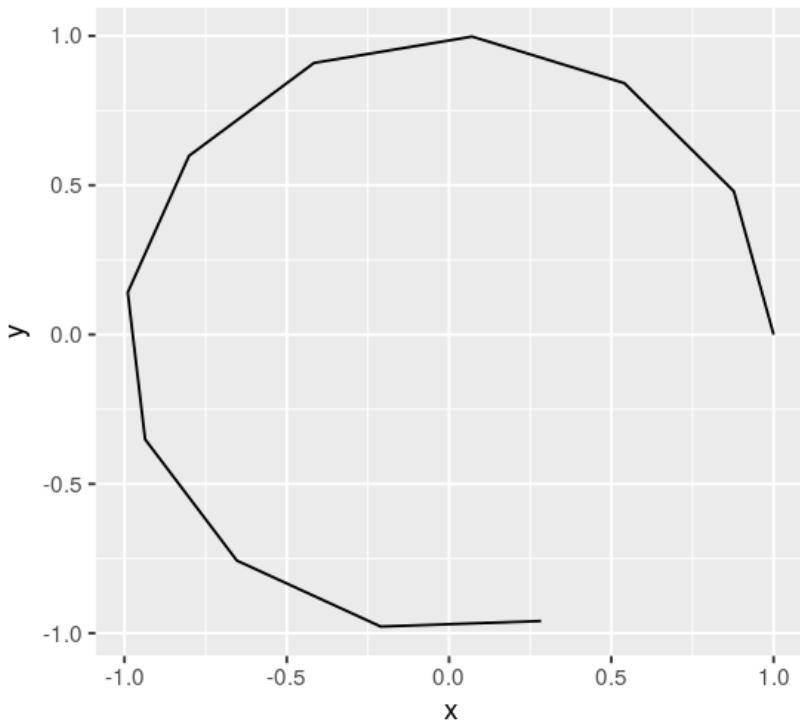
**Figure 9.17. This probably isn't what you want.**



The reason for this behaviour is that the points are joined as if they were sequentially produced along the x axis, which isn't the case for a circle. In this case, we can use an alternative; `geom_path()` which instead assumes the paired points are sequential, as in Figure 9.18.

```
ggplot(sin_df, aes(x, y)) + geom_path()
```

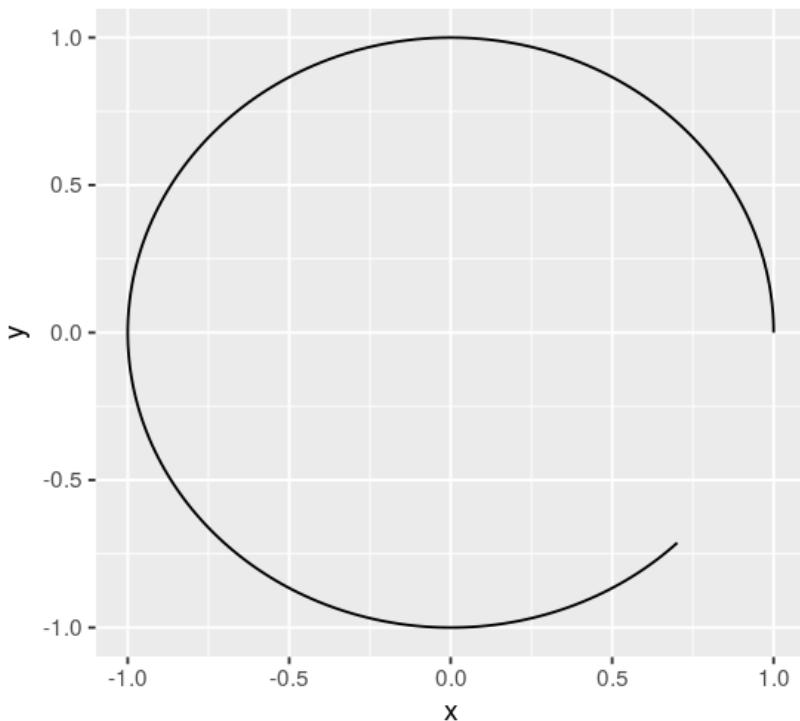
**Figure 9.18. Plotting the points as a 'path' assumes the pairs are to be plotted sequentially rather than in the order of the x values.**



If we make our data more finely scaled (with the `by` argument to `seq()`) we can even make this look more smoothly circular, as in Figure 9. 19.

```
sin_df <- data.frame(theta = seq(0, 1.75*pi, 0.01)) %>%
  mutate(x = cos(theta), y = sin(theta))
ggplot(sin_df, aes(x, y)) + geom_path()
```

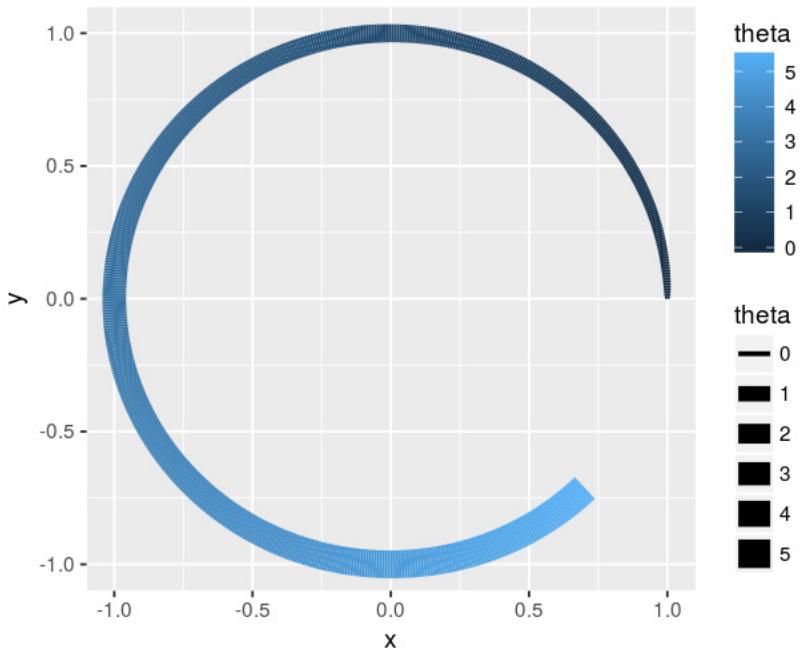
**Figure 9.19.** A more finely scaled data object produces a smoother curve when joining the points by lines.



We can apply some differential aesthetics to the path segments, such as line thickness (`lwd`) and colour (`col`), as in Figure 9.20.

```
ggplot(sin_df, aes(x, y)) + geom_path(aes(lwd = theta, col = theta))
```

**Figure 9.20.** Lines can be styled just as points can.



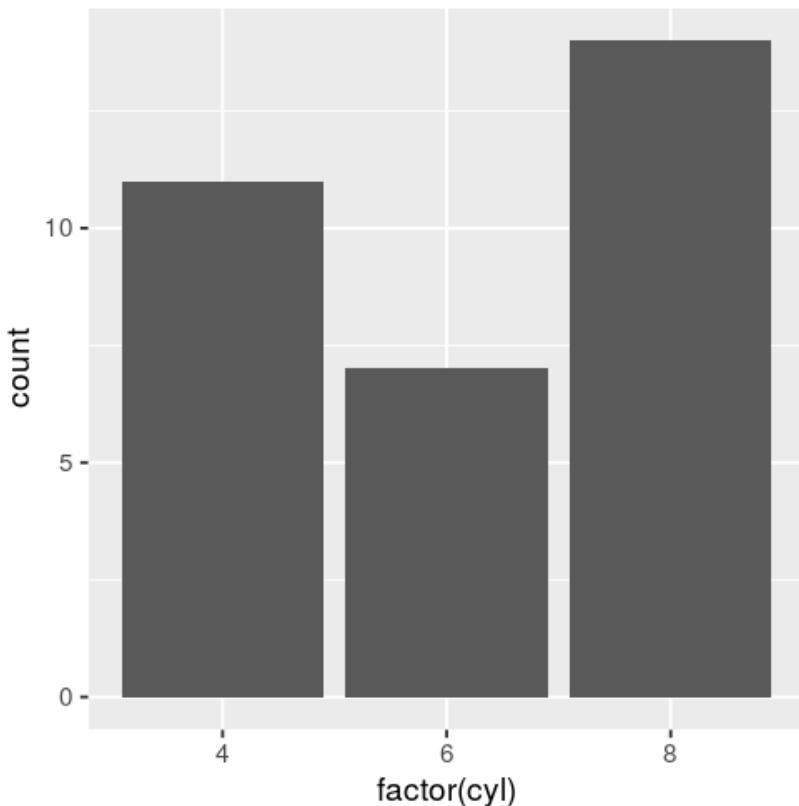
Note here that we *are* now taking advantage of the fact that `theta` isn't a factor, and the colour scale smoothly transitions from one colour to another.

### 9.2.5 Adding Bars

When we have categorical data such as counts of things, it's common to plot these as a barplot. One option would be to do the counting of the things yourself, but `ggplot2` is clever enough to do the work for you. If we wish to plot the number of vehicles in the `mtcars` dataset which have certain values of `cyl` (number of cylinders) then we can use `geom_bar()`, resulting in Figure 9. 21.

```
ggplot(mtcars) + geom_bar(aes(factor(cyl)))
```

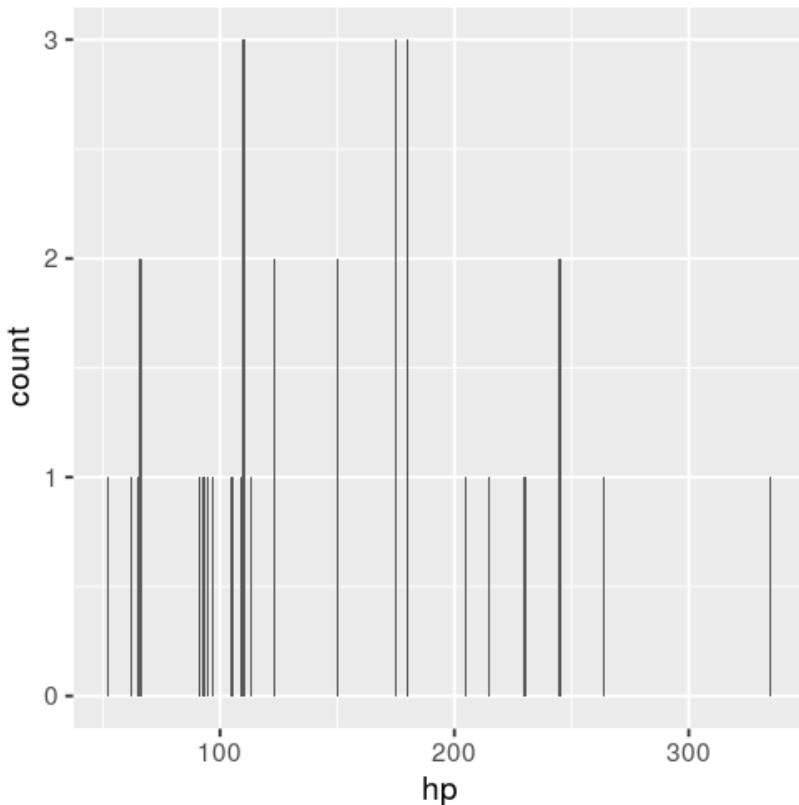
**Figure 9.21.** Categorical values can be counted and plotted as a barplot.



This works for any variable, even if it isn't so coarsely categorised, and even if it's continuous, such as Figure 9. 22.

```
ggplot(mtcars) + geom_bar(aes(hp))
```

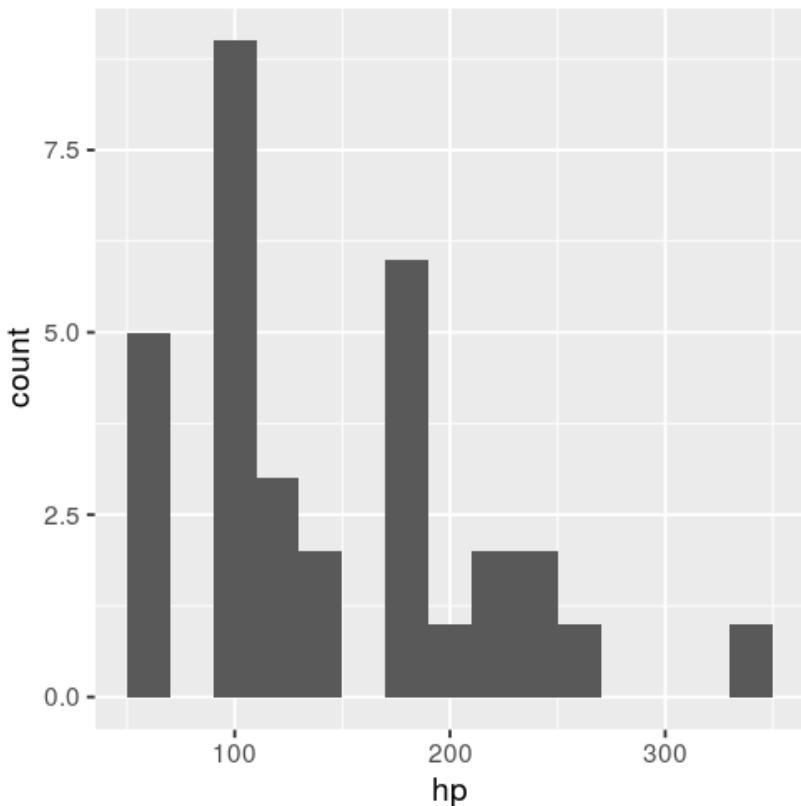
**Figure 9.22. Continuous values have counts in each distinct value.**



Here the counts of entries with a certain value of `hp` have been plotted as thin lines at that values on the x-axis. Note that we only need to specify the `x` argument; the count is generated within `geom_bar()` itself. Alternatively, we may consider these as individual groups with some width. If we group these `hp` values into bins 20 units wide, we can produce a histogram, such as in Figure 9.23.

```
ggplot(mtcars, aes(hp)) + geom_histogram(binwidth = 20)
```

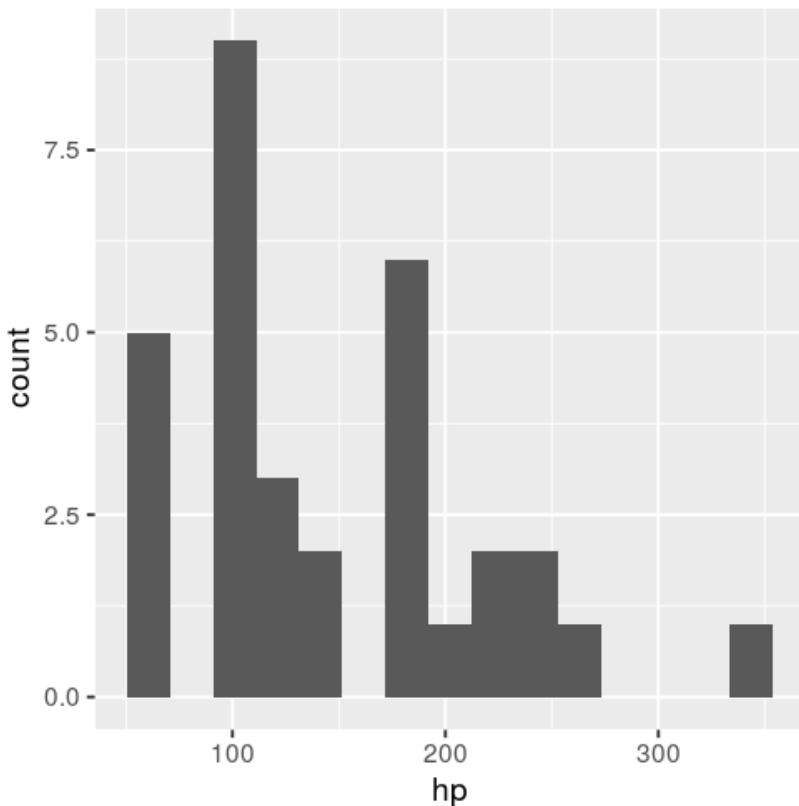
**Figure 9.23.** A histogram is a barplot with some binning.



which looks similar to the `geom_bar(aes(hp))` plot but with wider (20 units) bars. Rather than the width, we can select the number of bins into which the values will be grouped, such as Figure 9.24.

```
ggplot(mtcars, aes(hp)) + geom_histogram(bins = 15)
```

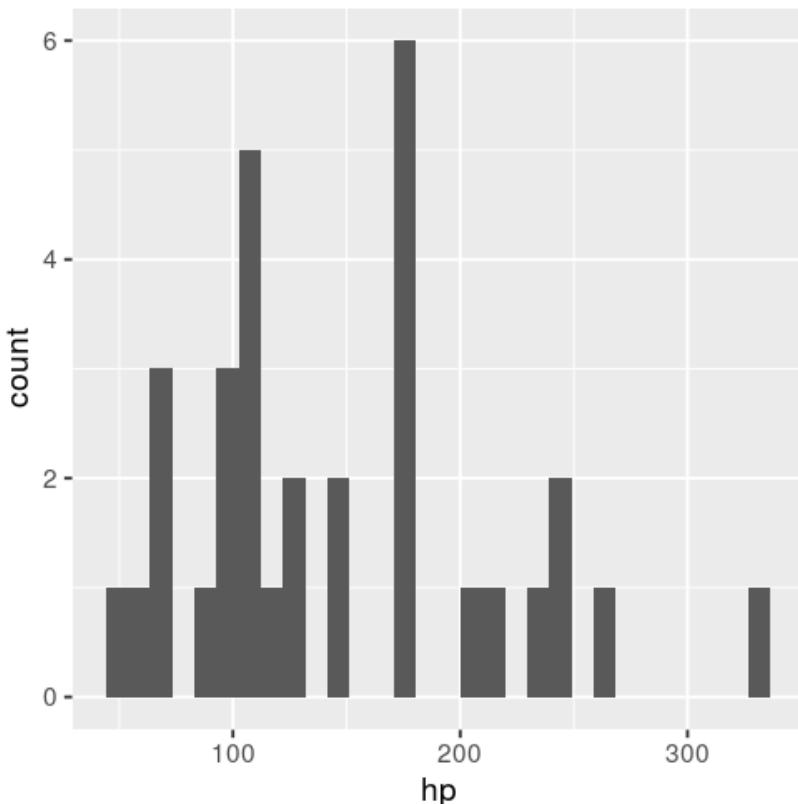
**Figure 9.24. Specifying 15 bins rather than the width of the bins.**



The value of `bins` has a default of 30 (Figure 9.25.) and this produces a message if you rely on it

```
ggplot(mtcars, aes(hp)) + geom_histogram()
# `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

**Figure 9.25. The default number of bins is 30.**



Another common scenario is to specify the heights of the bars yourself. If we manually count the number of vehicles with cyl cylinders (using our new `dplyr` skills), as well as the average miles per gallon (`mpg`) within those groups

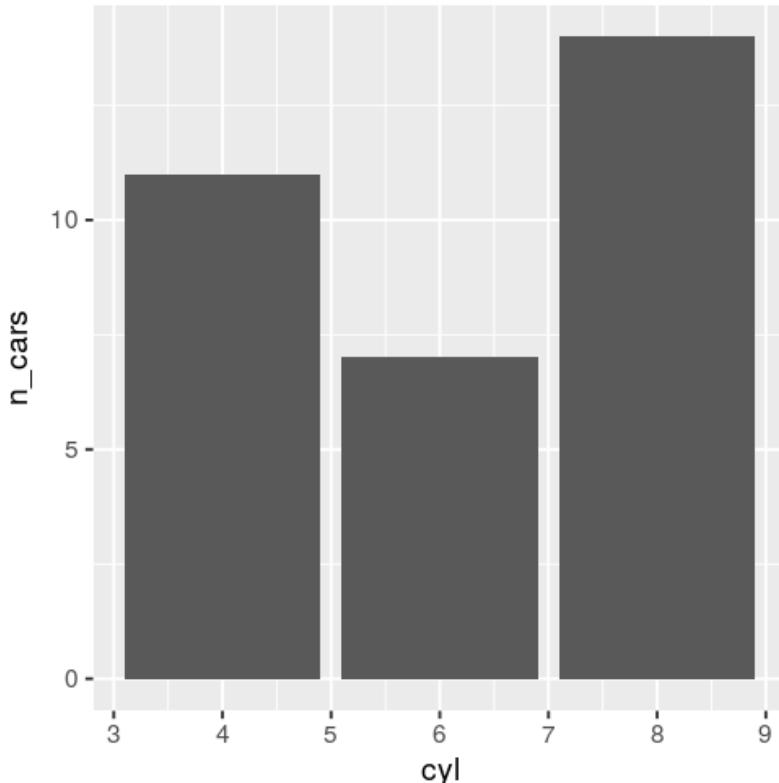
```
average_mpg_by_cyl <- mtcars %>%
  group_by(cyl) %>%
  summarise(n_cars = n(),
            avg_mpg = mean(mpg))
  )
```

- ➊ Group by number of cylinders, `cyl`.
- ➋ Calculate number of vehicles in each group, `n_cars` with the helper function `n()` which counts records within a group.
- ➌ Calculate the average miles per gallon in each group, `avg_mpg`.

then we can reproduce the `geom_bar()` result above manually (Figure 9.26.) if we specify that the statistic to run over this data is the `identity` transformation (no transformation)

```
ggplot(average_mpg_by_cyl, aes(cyl, n_cars)) + geom_bar(stat = "identity")
```

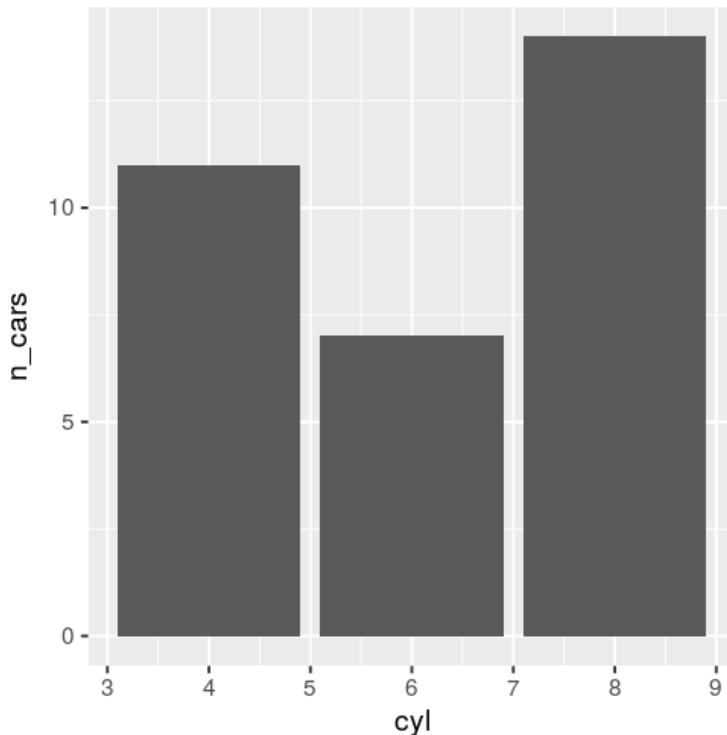
**Figure 9.26.** An 'identity' statistic indicates that counts do not need to be calculated.



There is a useful shortcut for this, the `geom_col()` which plots a 'column' with a specified height (Figure 9.27.).

```
ggplot(average_mpg_by_cyl, aes(cyl, n_cars)) + geom_col()
```

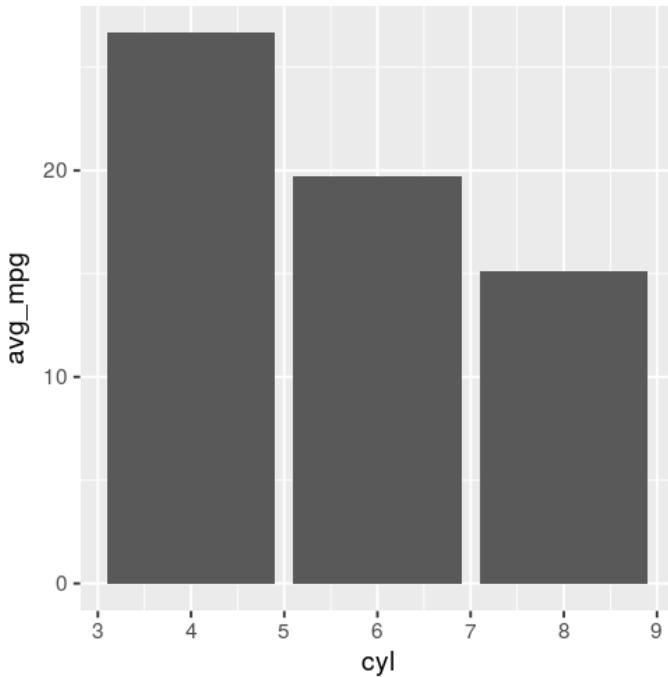
**Figure 9.27.** `geom\_col()` is useful if we know the heights of the bars.



In that case, we can plot any quantity for which we have calculated how tall we wish the bar to be (e.g. Figure 9.28.).

```
ggplot(average_mpg_by_cyl, aes(cyl, avg_mpg)) + geom_col()
```

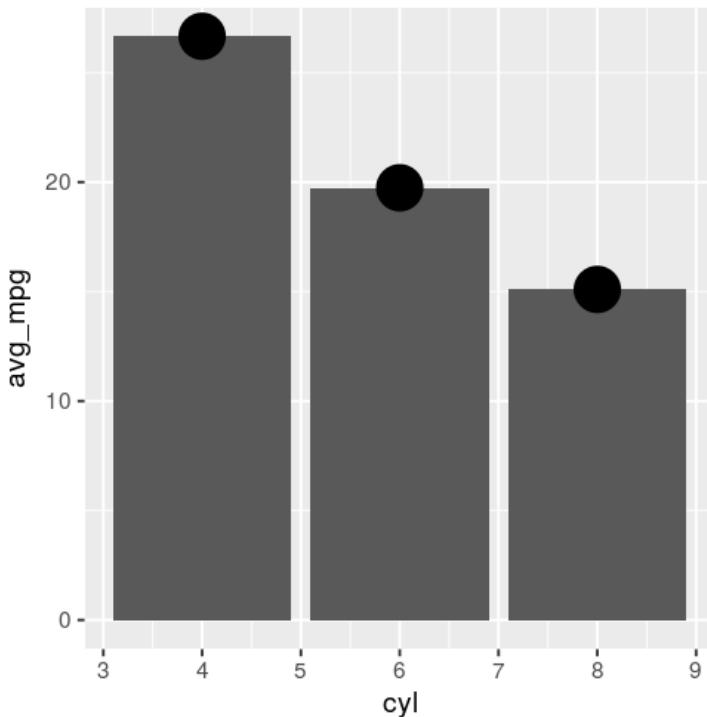
**Figure 9.28.** Another `geom_col()` plot.



We can combine this with other `geom_s` too. If we want the top of the bar to have a point on it, we can add that, making use of the inherited `aes()` (Figure 9.29.).

```
ggplot(average_mpg_by_cyl, aes(cyl, avg_mpg)) +  
  geom_col() +  
  geom_point(size = 8)
```

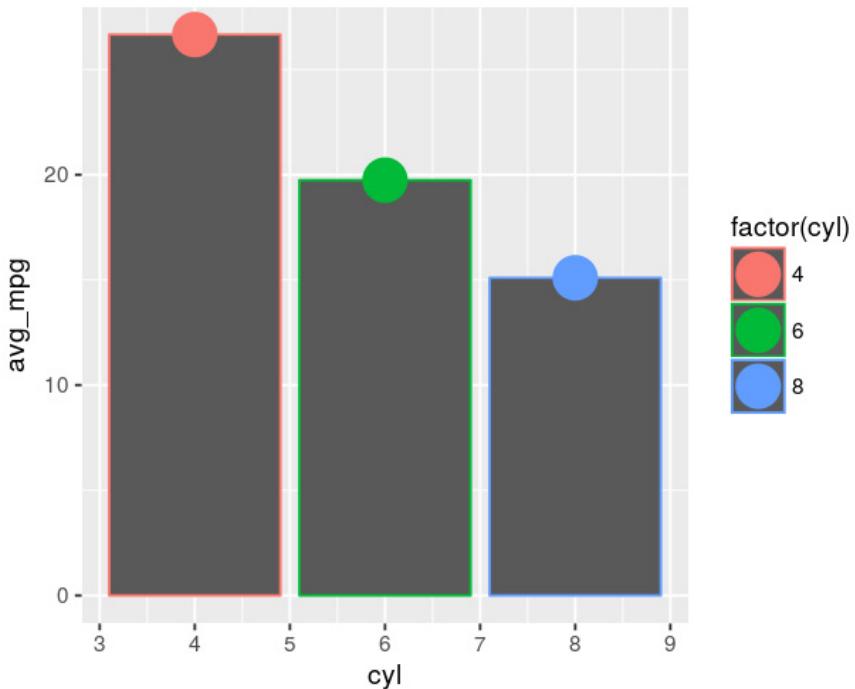
**Figure 9.29. Adding points to a geom\_col() plot.**



Again, colour can now be added in to more clearly delineate the categories. There are two options to do this. Trying to adapt our previous code using col actually changes the line colour for the `geom_col()`(Figure 9.30.).

```
ggplot(average_mpg_by_cyl, aes(cyl, avg_mpg, col = factor(cyl))) +
  geom_col() +
  geom_point(size = 8)
```

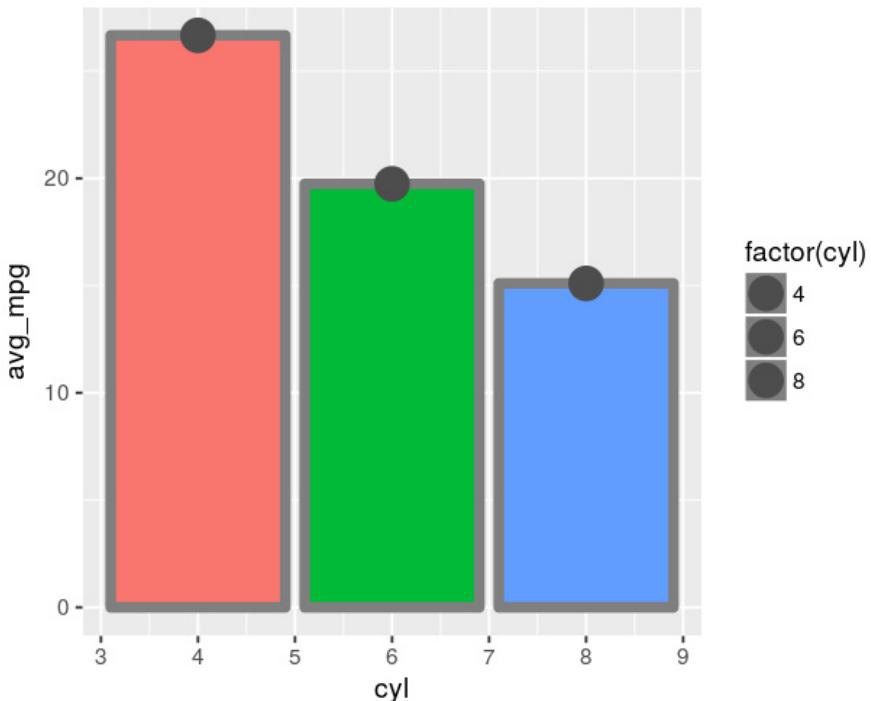
**Figure 9.30. The col aesthetic is inherited this changes the border line colour for geom\_col().**



To change the colour of the fill, we use the `fill` aesthetic. We can use this in tandem with the `col` aesthetic to produce some highlighting (e.g. Figure 9.31.).

```
ggplot(average_mpg_by_cyl, aes(cyl, avg_mpg, fill = factor(cyl))) +
  geom_col(col = "grey50", lwd = 2) +
  geom_point(col = "grey30", size = 6)
```

**Figure 9.31.** The `fill` aesthetic changes the internal colour of the bars.



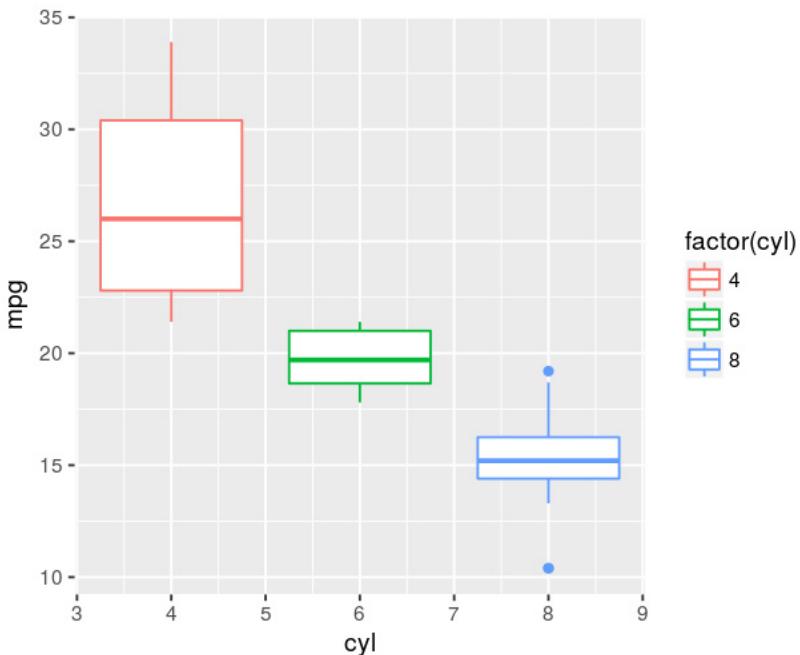
### 9.2.6 Other Types of Plots

To try to list all of the different types of plots made available by `ggplot2` and its extensions would require its own book (and several of those already exist). Instead, I'll point you in the direction of a few common and useful `geom_s`.

When plotting data which varies within a group, it can be useful to show the extent of that variability. A box-and-whiskers plot (a *boxplot*, Figure 9. 32.) can be useful in that case.

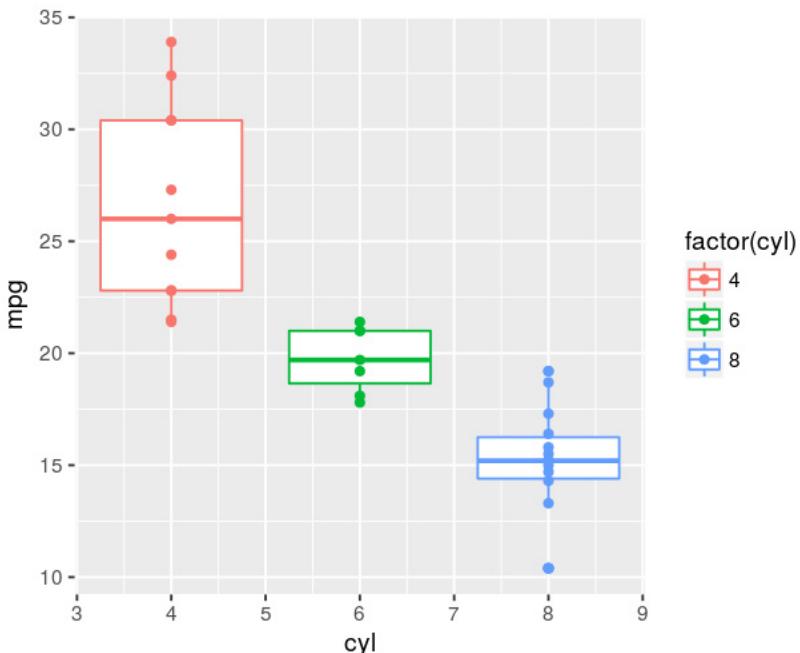
```
ggplot(mtcars, aes(cyl, mpg, col = factor(cyl))) +
  geom_boxplot()
```

**Figure 9.32.** A boxplot shows the data extent within a category.



The definitions for the edges of the boxes and the whiskers is found in `?geom_boxplot` but the interpretation is fairly standard. What this doesn't show (well) is the distribution of points in that range. We can add the points themselves if we wish (Figure 9.33.)

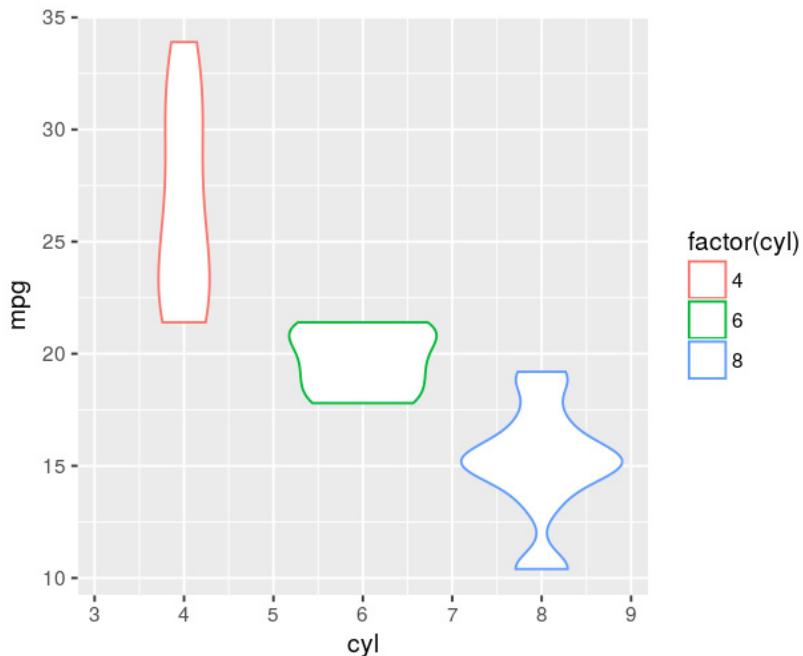
```
ggplot(mtcars, aes(cyl, mpg, col = factor(cyl))) +
  geom_boxplot() +
  geom_point()
```

**Figure 9.33. A boxplot with points overplotted.**

This is perhaps better handled by a *violin* plot (Figure 9.34.) which illustrates the density of points

```
ggplot(mtcars, aes(cyl, mpg, col = factor(cyl))) +
  geom_violin()
```

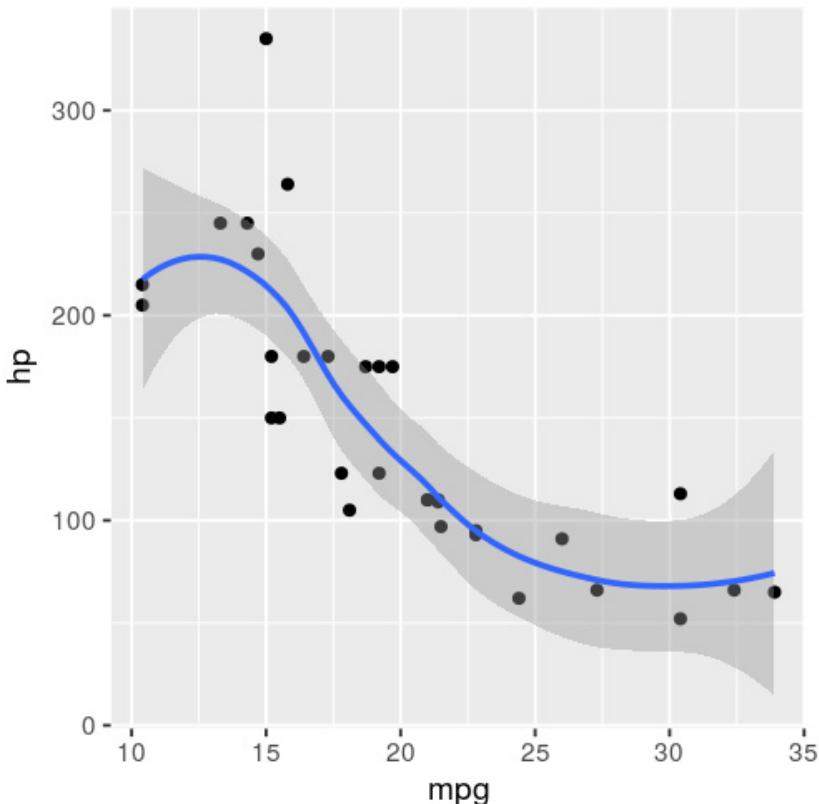
**Figure 9.34.** A violin extends the notion of a boxplot by showing the distribution of plots within each category.



We can show the general trend of some points (with or without the points themselves) using a *smoothing* function which calculates the local trend at any point (Figure 9.35.).

```
ggplot(mtcars, aes(mpg, hp)) +
  geom_point() +
  geom_smooth()
# `geom_smooth()` using method = 'loess'
```

**Figure 9.35.** A smoothing function shows the local trend at any point (with confidence interval shading).



The message produced tells us about the type of smoothing used and the assumed formula, and even though this is usually what we want, these can be overridden.

There are many ways to present data, and part of communicating it well is finding the right representation.

### 9.2.7 Scales

So far when we've asked `ggplot2` to handle data values as our aesthetics (e.g. `x = cyl` or `col = factor(am)`) we've trusted that it can decide how to do that. The range of the `x`-axis or the colours to be used are chosen by a default selection process. That won't always be what we want though.

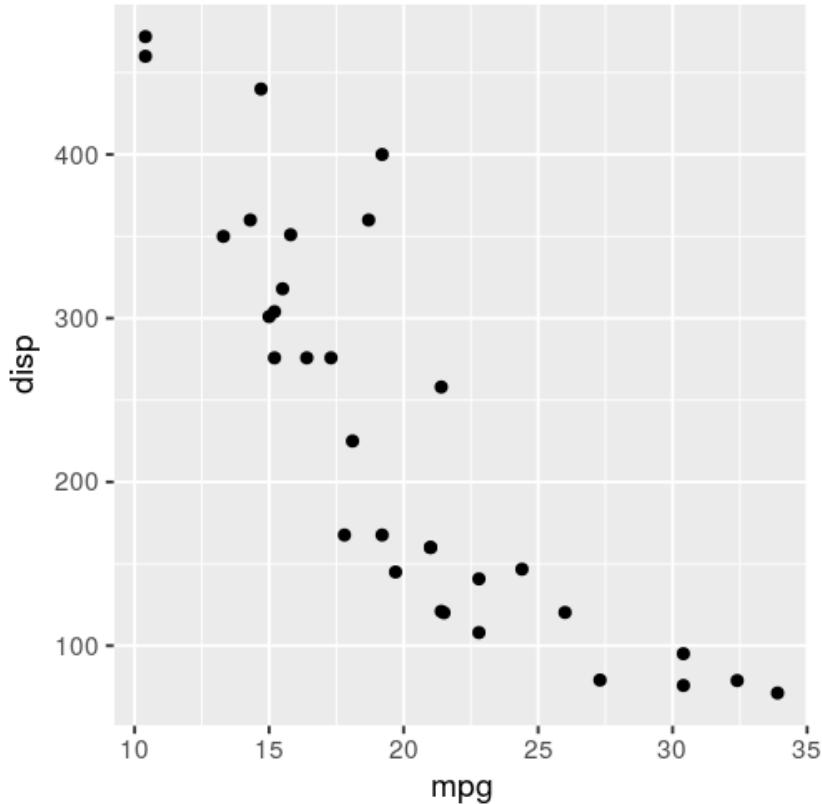
If we wish to change the range over which some feature is presented then we need the fourth ingredient in the 'grammar of graphics'; *scale*. These are accessed using `scale_` functions.

The simplest notion of changing a scale is the range of the `x`- or `y`-axis. This is

accomplished using either `scale_x_continuous()` or `scale_x_discrete()` depending on whether your data is of either of those definitions. You may have noticed that by default, `ggplot()` adjusts the axes ranges to roughly the extent of the data in each direction. Our go-to plot (Figure 9.36.) doesn't start at `x = 0`

```
ggplot(mtcars, aes(mpg, disp)) + geom_point()
```

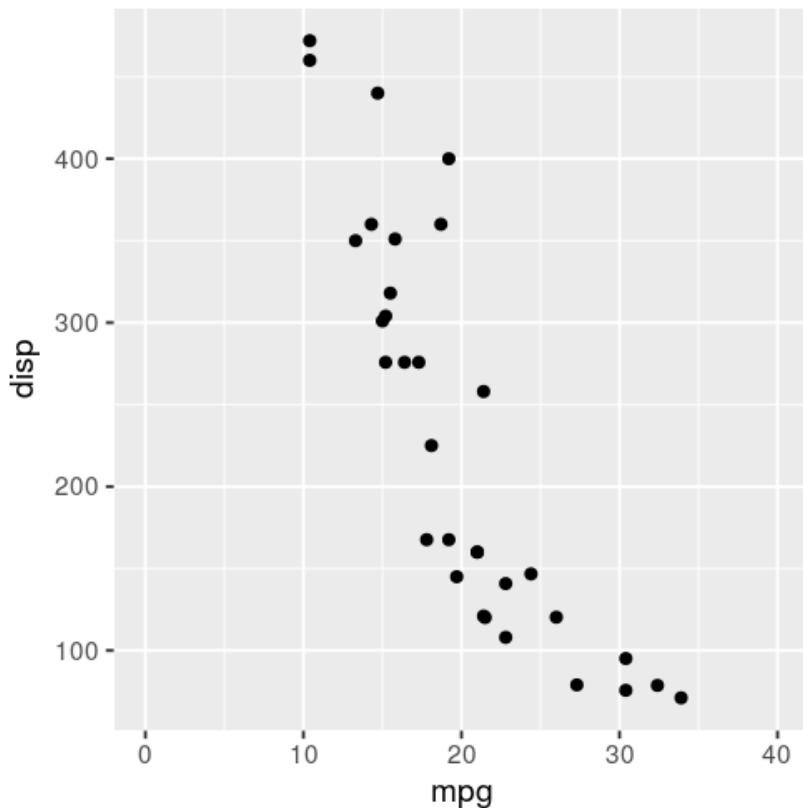
**Figure 9.36.** Note that the axes don't start at 0; they have been chosen to emphasise the existing data.



If we wish to enforce that (given that our x-values; `mpg` are continuous) we can set the limit of this using a vector which describes the start and end limits, as shown in Figure 9.37.

```
ggplot(mtcars, aes(mpg, disp)) +
  geom_point() +
  scale_x_continuous(limits = c(0, 40))
```

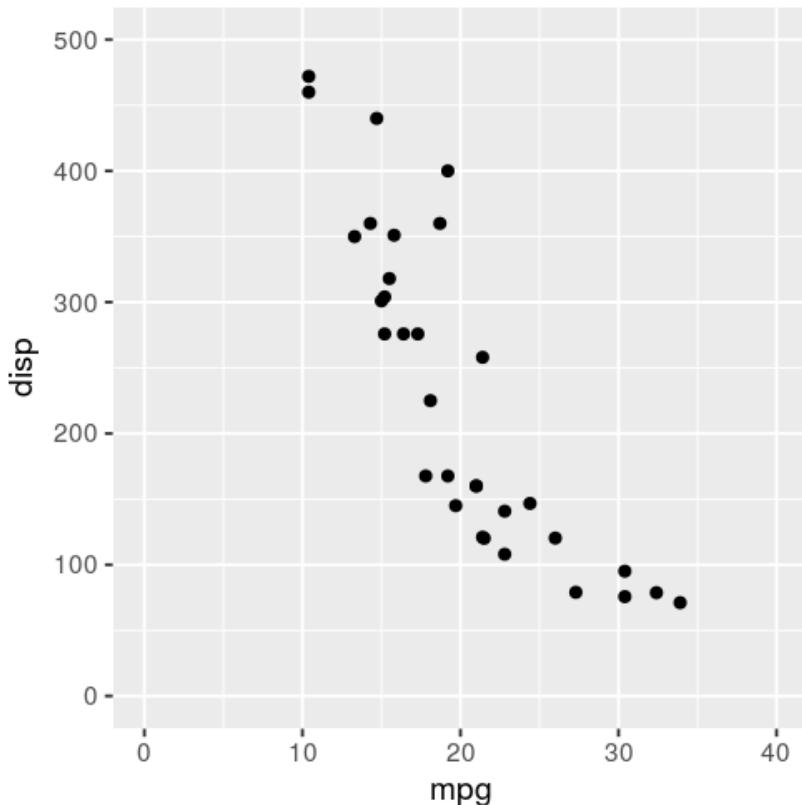
**Figure 9.37. Changing the x axis scaling with scale\_x\_continuous to start the range at 0.**



We can do the same for the y-axis (`disp`; also continuous) as shown in Figure 9.38.

```
ggplot(mtcars, aes(mpg, disp)) +
  geom_point() +
  scale_x_continuous(limits = c(0, 40)) +
  scale_y_continuous(limits = c(0, 500))
```

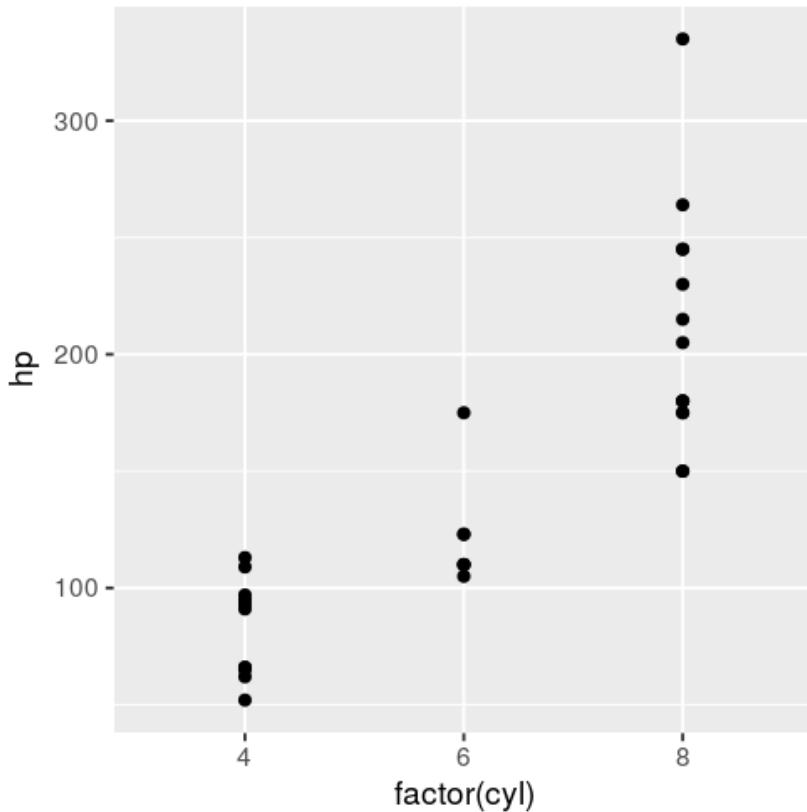
**Figure 9.38. Both axes can be scaled however you decide.**



When our data is discrete, we use an alternative `scale_` function. For example, if we treat `cyl` as a factor, we may wish to change the scale to include potential 2 cylinder vehicles (additional data on a Fiat 500 perhaps). Originally (Figure 9.39.) we have just the three factor levels present in the data

```
ggplot(mtcars, aes(factor(cyl), hp)) +
  geom_point()
```

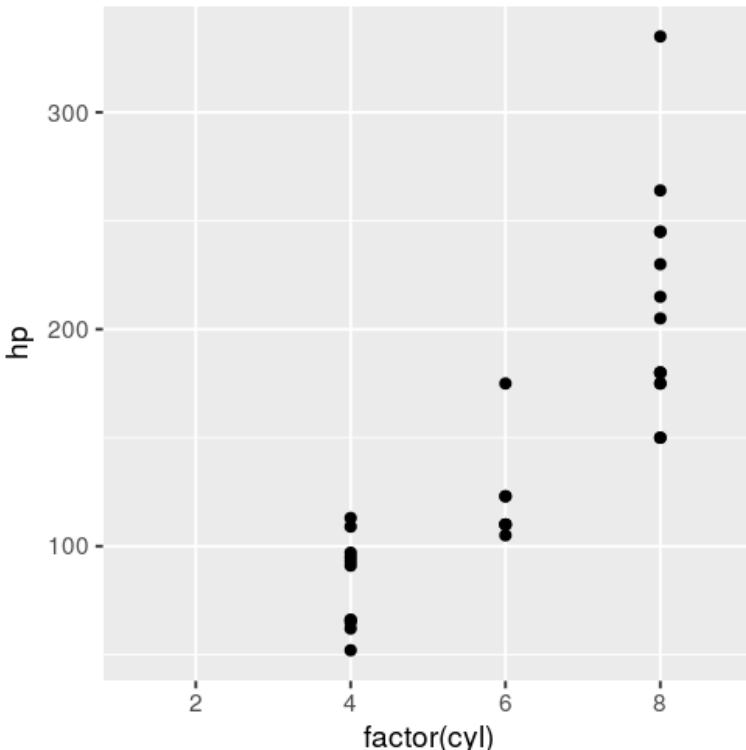
**Figure 9.39.** This shows the three existing levels in cyl.



But we can extend that, as has been done in figure 9.40.

```
ggplot(mtcars, aes(factor(cyl), hp)) +
  geom_point() +
  scale_x_discrete(limits = as.character(seq(2, 8, by = 2))) ①
```

**Figure 9.40.** We can extend the scale to a not-yet-present fourth level.



- In this example the categories are factor levels, so our limits also need to be strings.

This usage of `limits` differs somewhat from the usage in `scale_x_continuous()`. The help file for `scale_x_discrete()` notes that the meaning of the argument `limits` is

*“ limits: A character vector that defines possible values of the scale and their order.”*

---

-- Help file for `scale_x_discrete()`

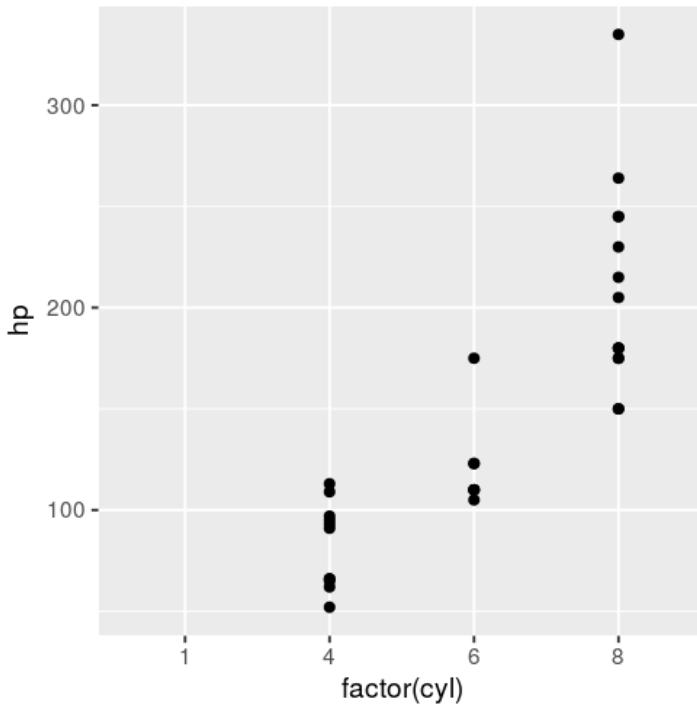
That is what I have provided above; a character vector (due to `as.character()`) of the values I wish to be used on the x-axis, and their order. These are `c("2", "4", "6", "8")`.



The spacing of factors on an axis is, by default, even. Our example of counting `cyl` by twos means that it looks clean on the axis, but if we have an uneven spacing of factor levels, say, if the first level was "1" and not "2", as in [Figure 9.41](#).

```
ggplot(mtcars, aes(factor(cyl), hp)) +
  geom_point() +
  scale_x_discrete(limits = as.character(c(1, 4, 6, 8)))
```

**Figure 9.41.** Categories appear at even spacing along the axis by default.



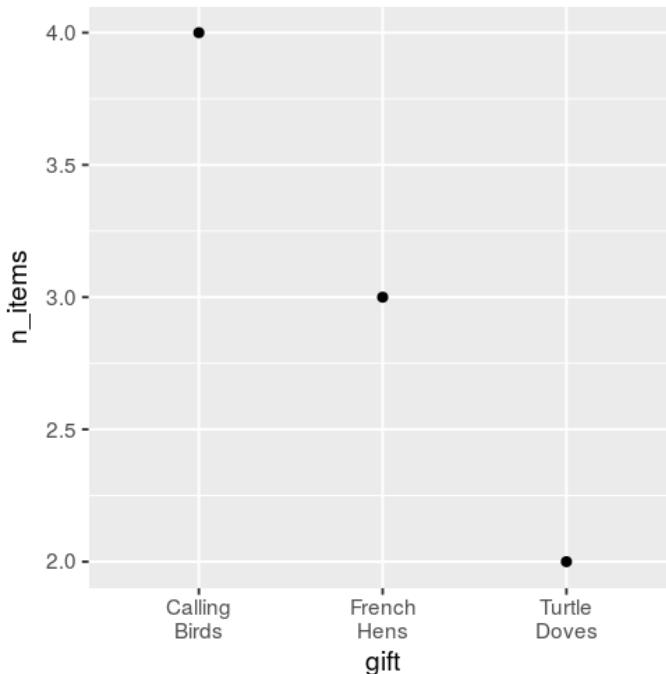
Then it looks less clean. Remember though; we are defining these as categories, not as numbers, so the spacing between them should be equal. Similarly, if our categories didn't resemble numbers, it would make just as much sense to have them equally spaced. If we have some data with three factor levels

```
gifts <- data.frame(
  gift = c("French\nHens", "Turtle\nDoves", "Calling\nBirds"), ①
  n_items = c(3, 2, 4)
)
```

① Remember, factor levels can contain escaped characters; see what `ggplot()` does with them.

we can plot them with the default range (Figure 9. 42.).

```
ggplot(gifts, aes(gift, n_items)) +
  geom_point()
```

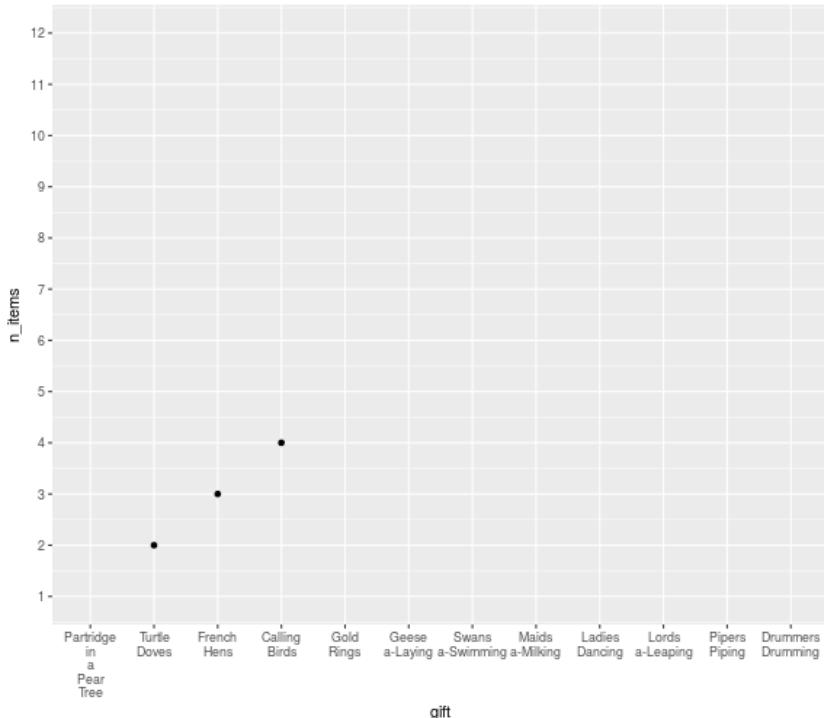
**Figure 9.42. Three categories (factor levels) plotted.**

This is just the data for that small subset of what will eventually be all of the data. Rather than creating 'empty' data in our 'clean' dataset, we can specify that the limits simply need to be expanded by providing the full list of options, resulting in Figure 9.43.

```
all_gifts <- c(
  "Partridge\nin\na\nPear\nTree",
  "Turtle\nDoves",
  "French\nHens",
  "Calling\nBirds",
  "Gold\nRings",
  "Geese\nLaying",
  "Swans\nna-Swimming",
  "Maids\nna-Milking",
  "Ladies\nDancing",
  "Lords\nna-Leaping",
  "Pipers\nPiping",
  "Drummers\nDrumming"
)

ggplot(gifts, aes(gift, n_items)) +
  geom_point() +
  scale_x_discrete(limits = all_gifts) +
  scale_y_continuous(limits = c(1, 12), breaks = 1:12) ① ②
```

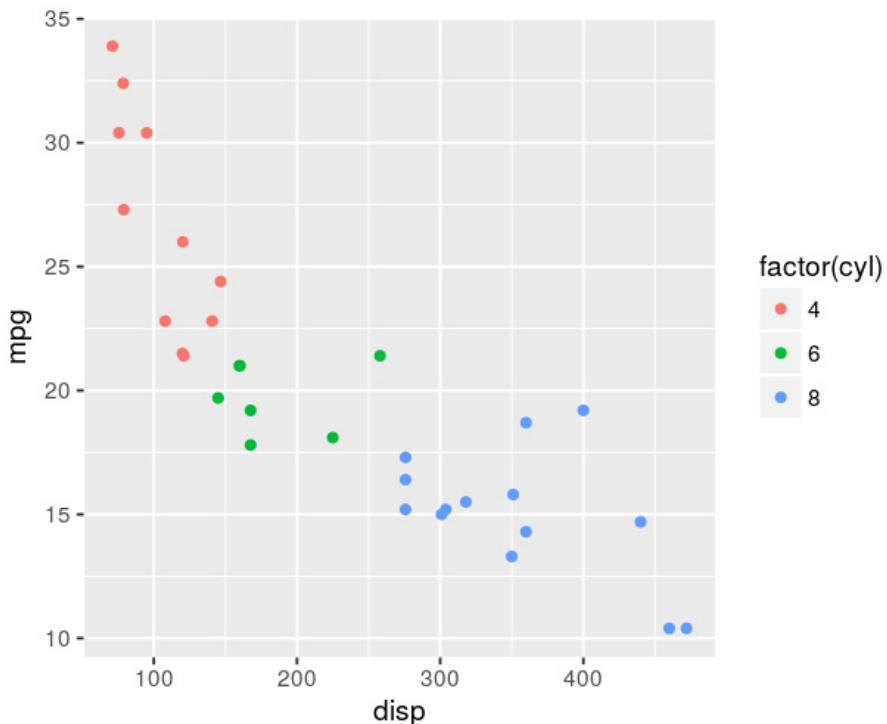
**Figure 9.43.** We can provide more levels in the scale even if they don't yet exist in the data.



- ➊ The levels, and their ordering, have been provided using the `all_gifts` vector above.
- ➋ I have used the argument `breaks` to specify that I don't want fractional tick marks on the y-axis.

The axes aren't the only scale we can work with; any of the aesthetics potentially has a scale we can manipulate. The default set of colours that `ggplot()` uses aren't the only options; returning to our plot (Figure 9.44.) for which `col` was specified by the level of `cyl`

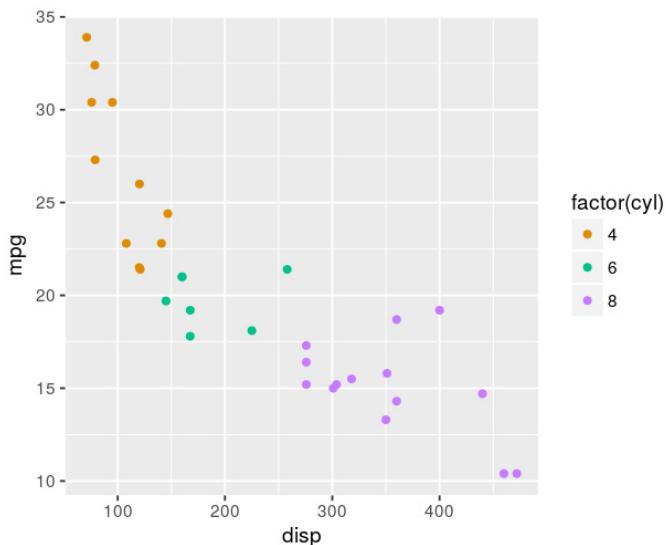
```
ggplot(mtcars, aes(disp, mpg, col = factor(cyl))) +
  geom_point()
```

**Figure 9.44. Default colours for three levels.**

If we want those colours to be different, we can use `scale_color_discrete()` (or `scale_colour_continuous()`) function. We could ask for the three colours to be chosen from the colour wheel, but starting at a different point, by specifying the starting angle. Say, 30 degrees, resulting in Figure 9.45.

```
ggplot(mtcars, aes(disp, mpg, col = factor(cyl))) +
  geom_point() +
  scale_color_discrete(h.start = 30) ❶
```

**Figure 9.45. Starting the colour wheel at 30 degrees results in different colours.**

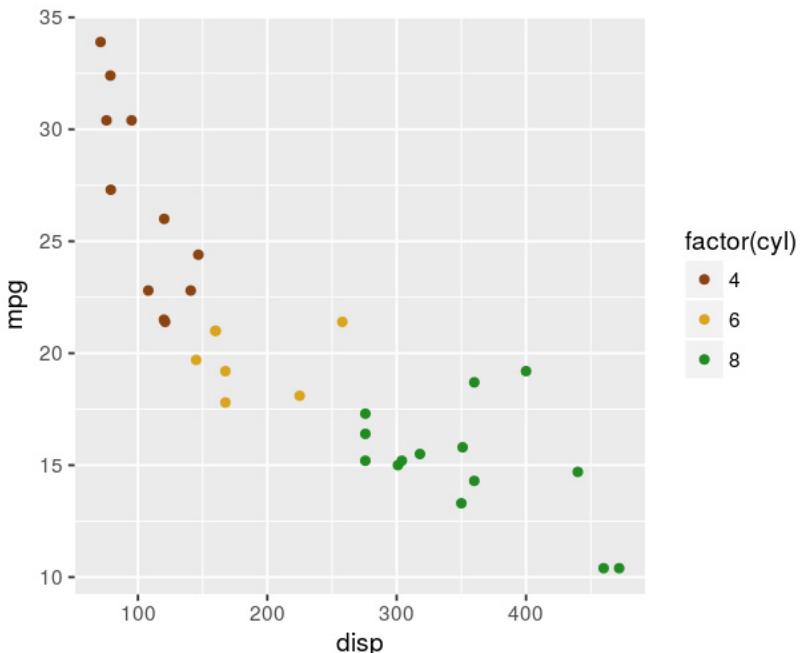


- ➊ The default starting angle is 15 degrees.

Or we could specify the colours manually, as in Figure 9.46.

```
ggplot(mtcars, aes(disp, mpg, col = factor(cyl))) +
  geom_point() +
  scale_color_manual(values = c("saddlebrown", "goldenrod", "forestgreen"))
```

**Figure 9.46.** We can manually choose colours too.



There are many palettes of colours available. Read `?scale_color_manual` for many examples.

There are similar `scale_` functions for most of the common aesthetics, and you should become familiar with as many of them as you can.

### 9.2.8 Facetting

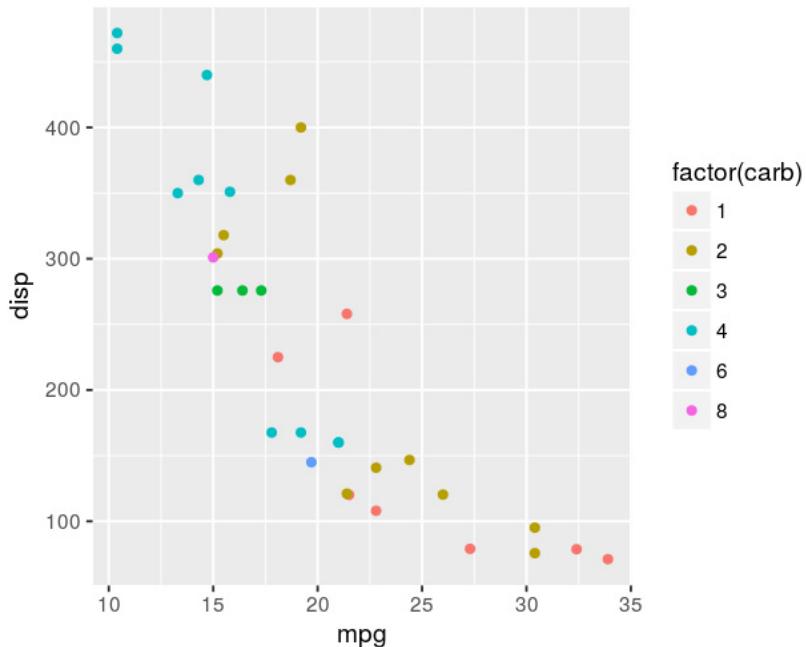
We've seen that we can use aesthetics to distinguish characteristics but this isn't always clear enough, especially if points or lines overlap considerably between groups. There is one more powerful feature of `ggplot2` we can use to physically separate features based on some quantity, and that is *facetting*.

Rather than simply changing an aesthetic of each group, we can create individual plots for each group where only those values will be plotted. This is all handled automatically for us, so it's as simple as adding a call to `facet_grid()` or `facet_wrap()`, depending on how we wish to layout the individual plots.

If we look at the values of `mpg` as they vary by `disp`, we can colour them according to their value of `carb` resulting in Figure 9.47.

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +
  geom_point()
```

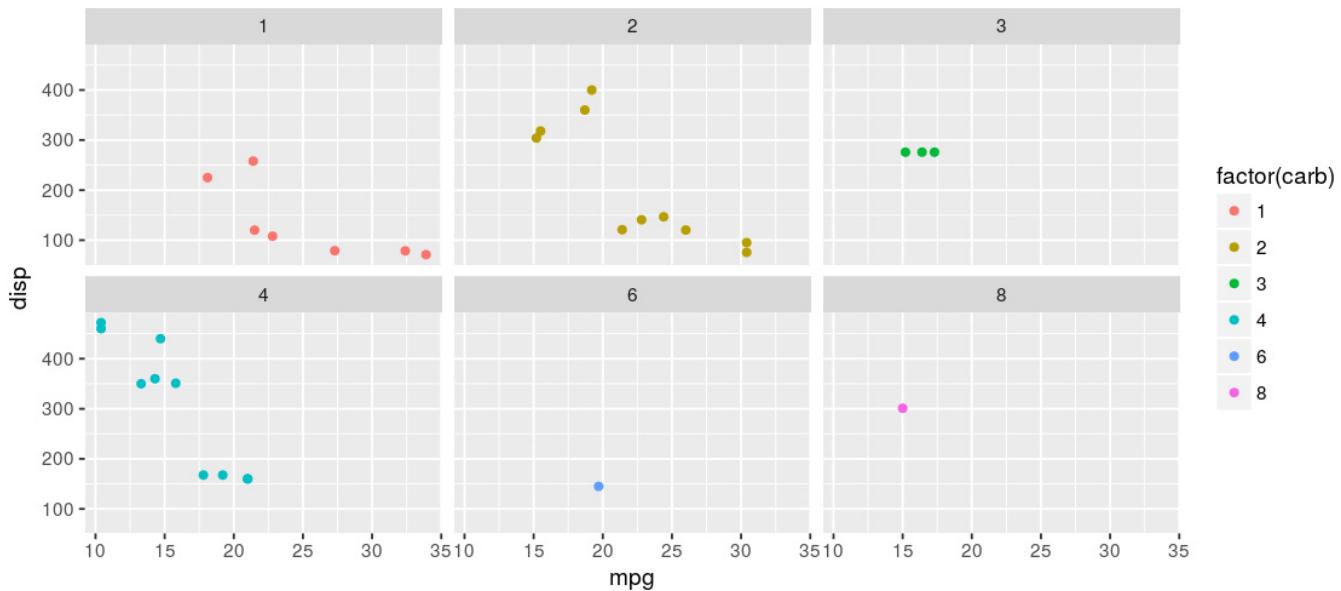
**Figure 9.47. One way to show another dimension is to use colour to distinguish levels of a group.**



but this category grouping isn't very clear; it's difficult to see what the relationship is between mpg and disp for each group. Instead, we can physically separate the different levels of carb onto their own plots, as in Figure 9.48.

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +  
  geom_point() +  
  facet_wrap(~ carb)
```

**Figure 9.48. Facetting moves each category level to its own plot.**



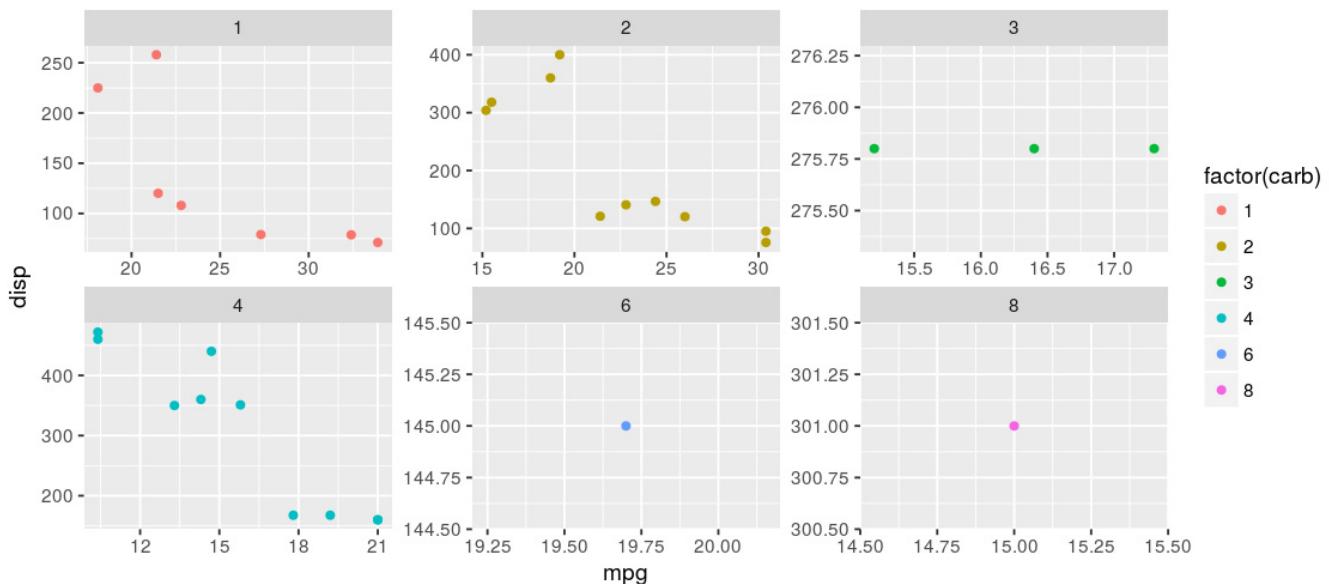
The syntax for facetting is that of a *formula*. For `facet_wrap()` a single-sided formula produces the effect we want. Equivalently, we can provide the argument as a character vector, so `facet_wrap("carb")` would produce the same result.

Note that each facet is labelled in the *strip* by the value of carb (values 1 through 8, missing 5 and 7) for that group. Individual relationships within that group are now clearer.

Some of the groups don't have many values, so it's sometimes useful to be able to change the scale. We need to change it differently for each facet though, so that option is built right in as the `scales` argument, which takes a "free" value to make the scaling independent for each facet (Figure 9.49.).

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +
  geom_point() +
  facet_wrap(~ carb, scales = "free")
```

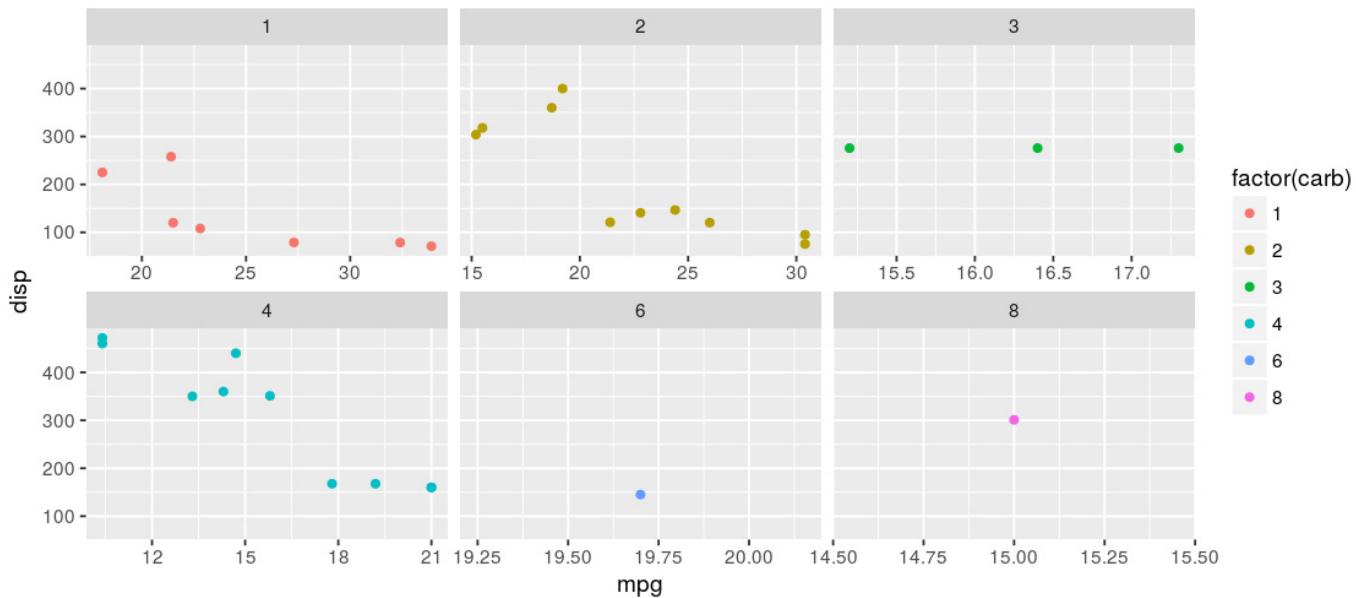
**Figure 9.49. Using `scales = "free" means the scale in each facet panel is independent.**



The options for the argument `scales` are "fixed" (common across facets); "free" (freely varying across facets); or "free\_x" or "free\_y" (freely varying in just one dimension). If we want to keep the y-axis on a common scale but allow the x-axis to be scaled differently, that's simple enough, as in Figure 9.50.

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +
  geom_point() +
  facet_wrap(~ carb, scales = "free_x")
```

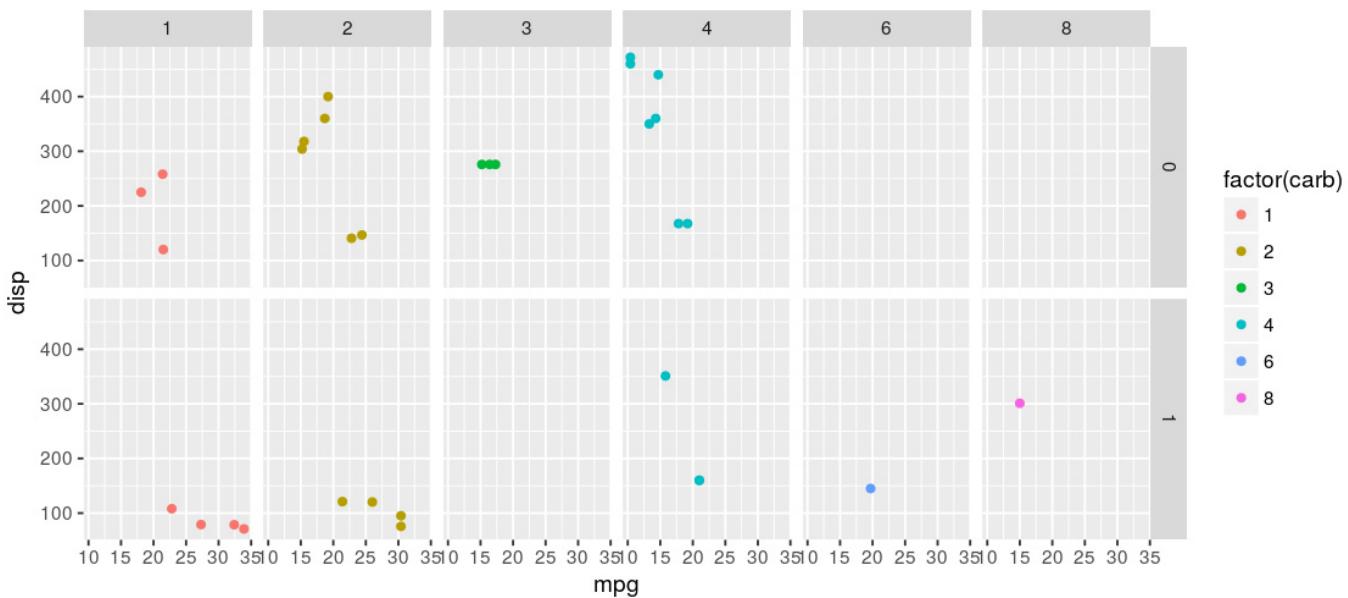
**Figure 9.50.** We can keep only one axis fixed using scales = "free\_x".



The other faceting function is `facet_grid()` which combines two different features in a grid of facets. For example, if we think that the relationship between `mpg` and `disp` may behave differently depending on both `carb` and `am`, we can investigate that and produce Figure 9.51.

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +
  geom_point() +
  facet_grid(am ~ carb)
```

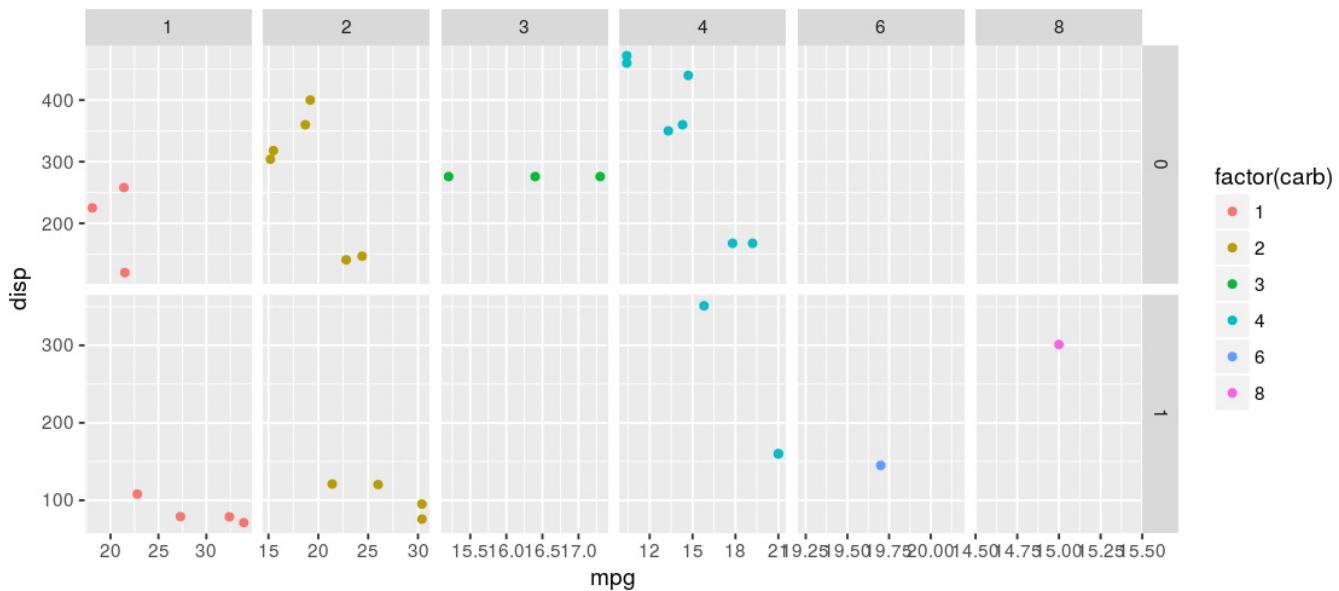
**Figure 9.51.** We can make comparisons between two levels in a grid of values using `facet_grid()`.



Here the value of `carb` varies across a row of facets, while `am` varies between the columns of facets. Again, if we wish to allow the scales to vary, we can add that in and produce Figure 9.52.

```
ggplot(mtcars, aes(mpg, disp, col = factor(carb))) +
  geom_point() +
  facet_grid(am ~ carb, scales = "free")
```

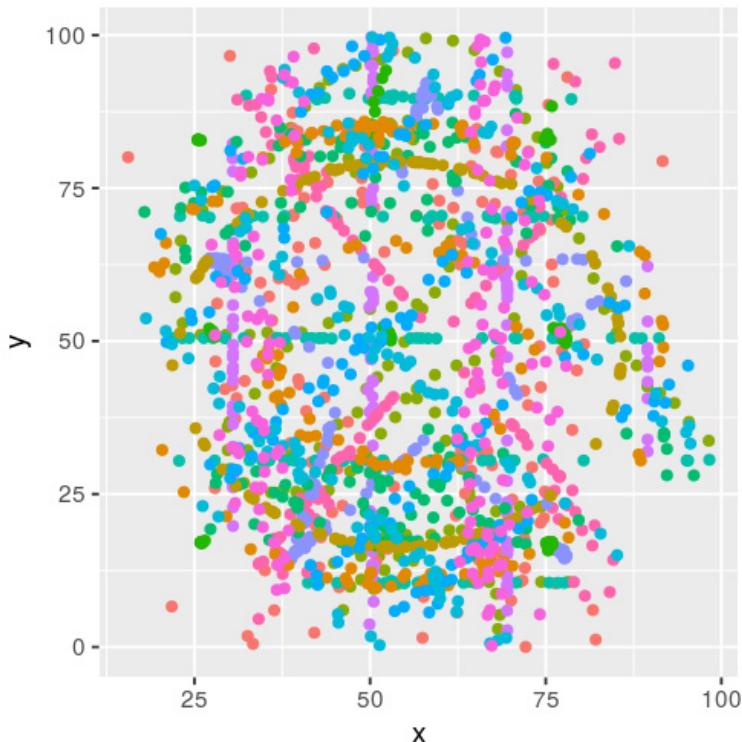
**Figure 9.52. Using the scales argument to make the axes independent.**



Facets at the intersections of valid values of `carb` and `am` with no data (e.g. there is no data for `carb == 3` and `am == 1`) will still be shown when presented this way (the layout will always be a grid).

This is an extremely powerful tool which can give us insight into hidden patterns not otherwise visible. When you look at the overlap of a dozen datasets (coloured by the dataset to which they belong) it's impossible to tease out any patterns (see Figure 9.53.).

**Figure 9.53.** Can you see the patterns in this plot? There are 13 different datasets.

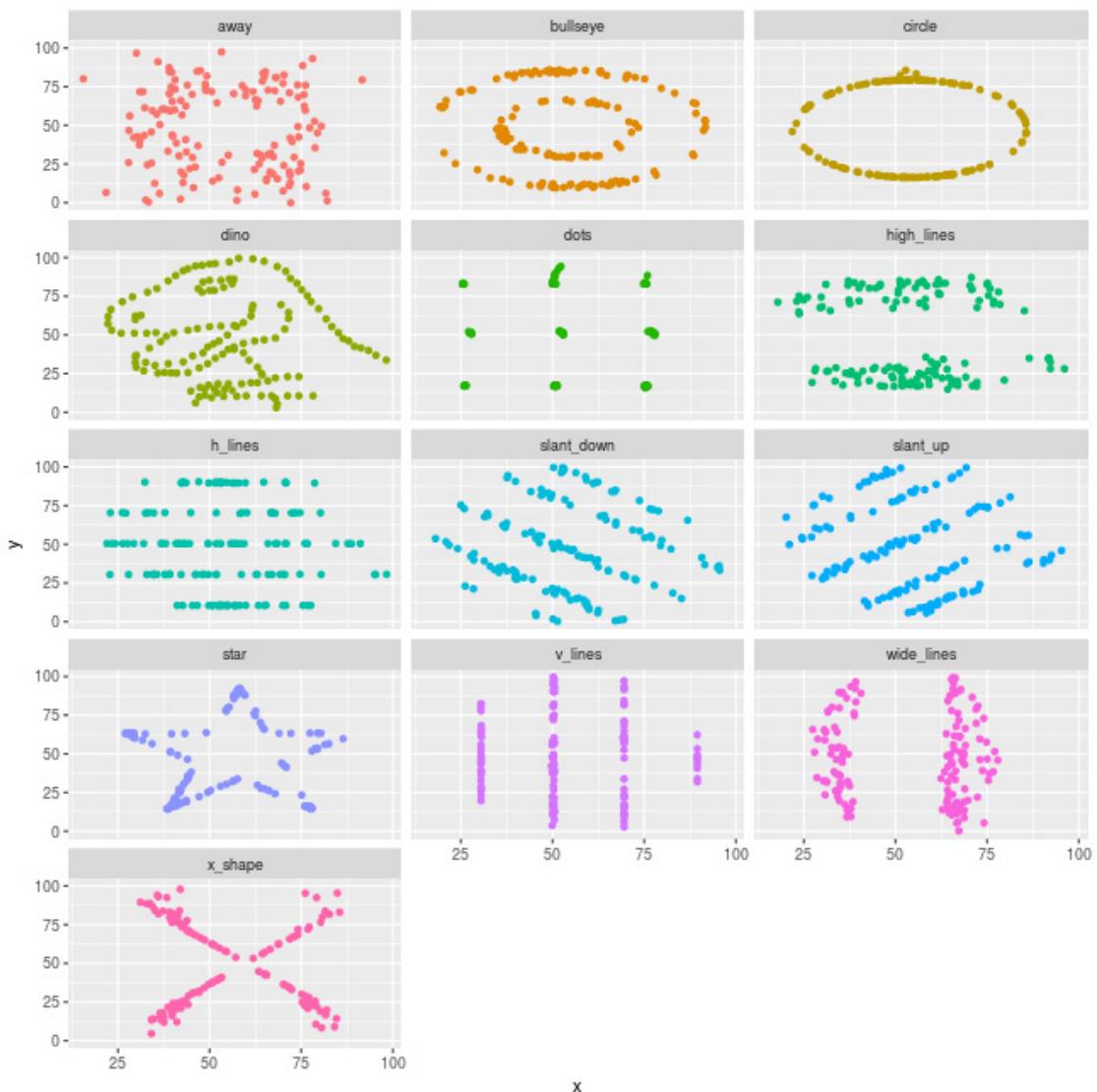


Facet the data by that grouping however, and it's an entirely different story.<sup>72</sup> Both Figure 9.53. and Figure 9.54. are plotting the same data, but faceting helps to show the differences in a way that colour alone could not do.

---

<sup>72</sup> This dataset as a whole is the 'datasaurus dozen' and is captured in the `datasaurRus` package by Steph Locke and Lucy D'Agostino McGowan. Each individual grouping of data shares the same mean and standard deviation for both the x and y values, as well as the same correlation between those.

**Figure 9.54.** The datasets are revealed with facetting.



### 9.2.9 Additional Options

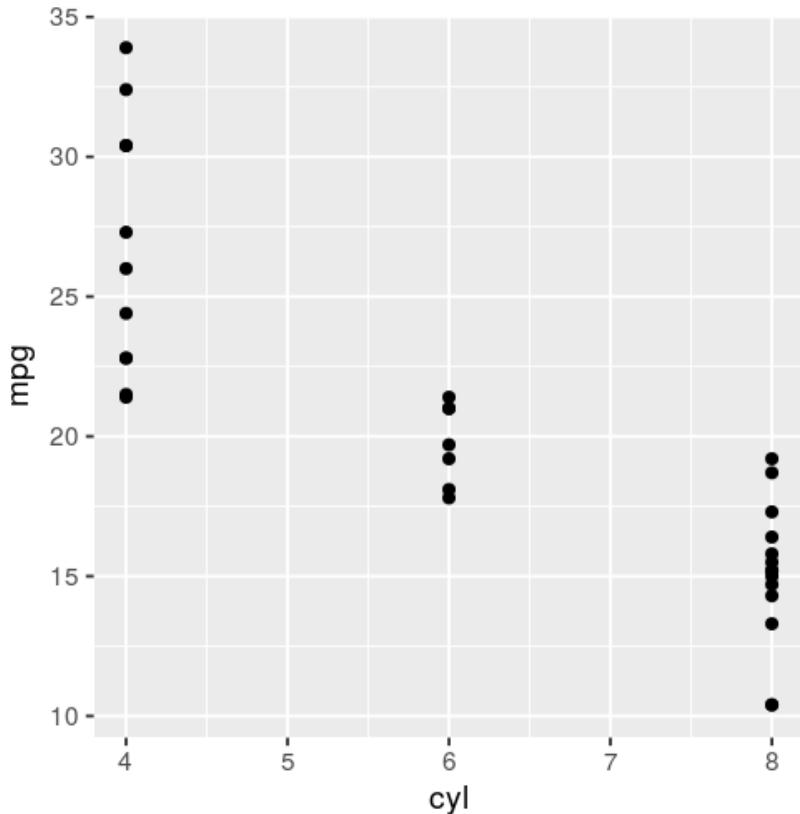
All of the plots we've generated so far have had the same default *theme*; the grey background with gridlines. This is only the default however, and it is entirely customisable. The `theme()` function takes a wide variety of arguments to specify everything from the background to the fonts to the way the axis labels are shown, and so on.

Since there are so many options, various `theme_` functions exist to specify certain

combinations. The default is actually `theme_grey()`. For example, the default plot we started with looks like (Figure 9. 55.)

```
ggplot(mtcars, aes(cyl, mpg)) +
  geom_point()
```

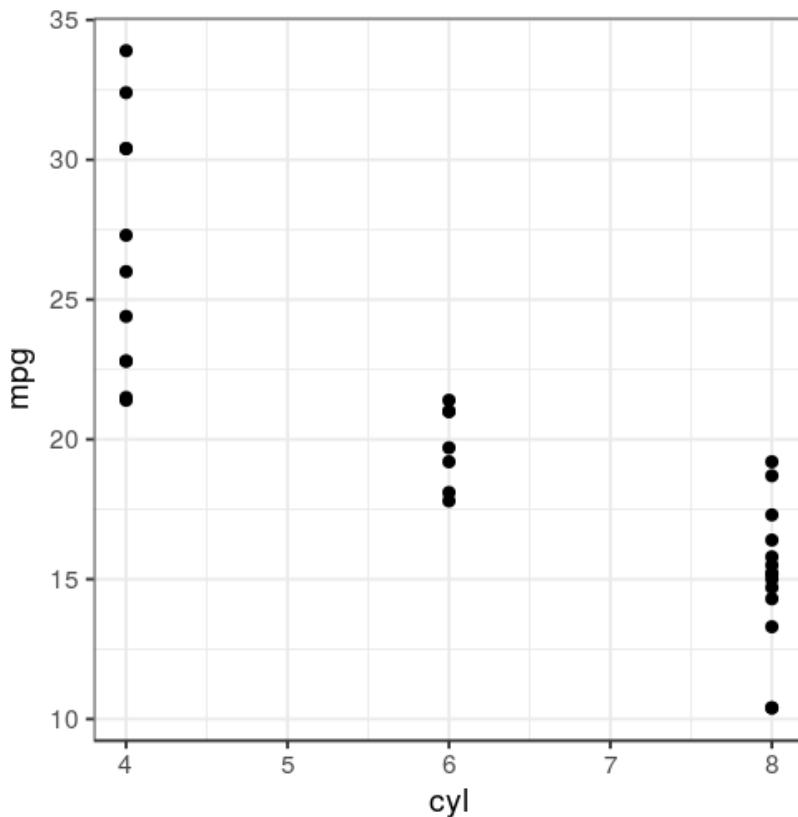
**Figure 9.55. The default theme for ggplot2 plots in action.**



but we can change the theme to a more black-and-white version with `theme_bw()`, as shown in Figure 9. 56.

```
ggplot(mtcars, aes(cyl, mpg)) +
  geom_point() +
  theme_bw()
```

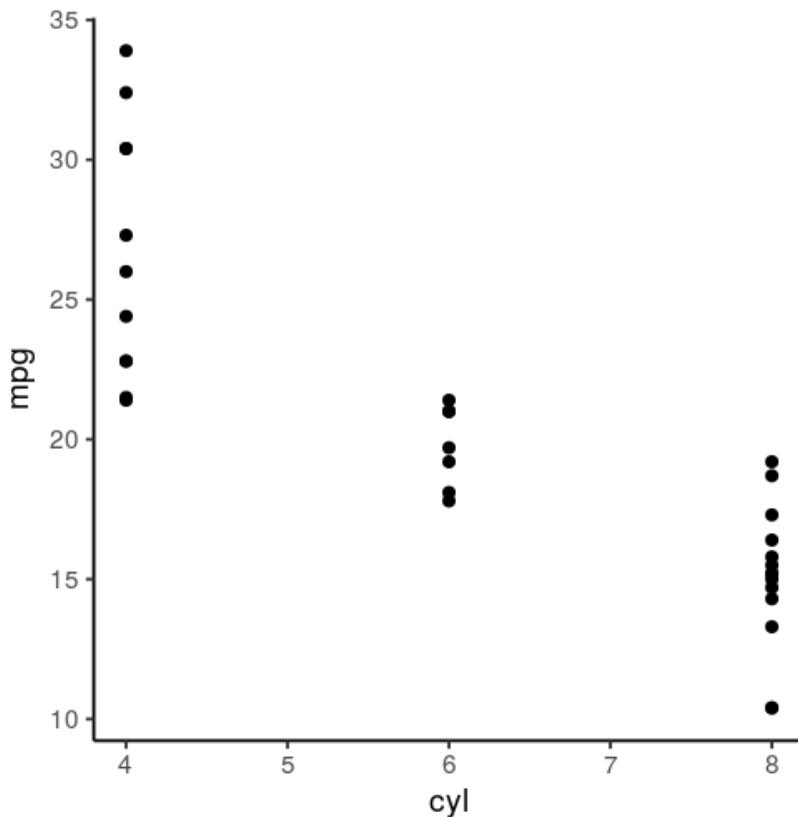
**Figure 9.56.** `theme_bw()` is a bit clearer and less styled than the default.



We can remove all but the essentials with `theme_classic()` (Figure 9.57.).

```
ggplot(mtcars, aes(cyl, mpg)) +
  geom_point() +
  theme_classic()
```

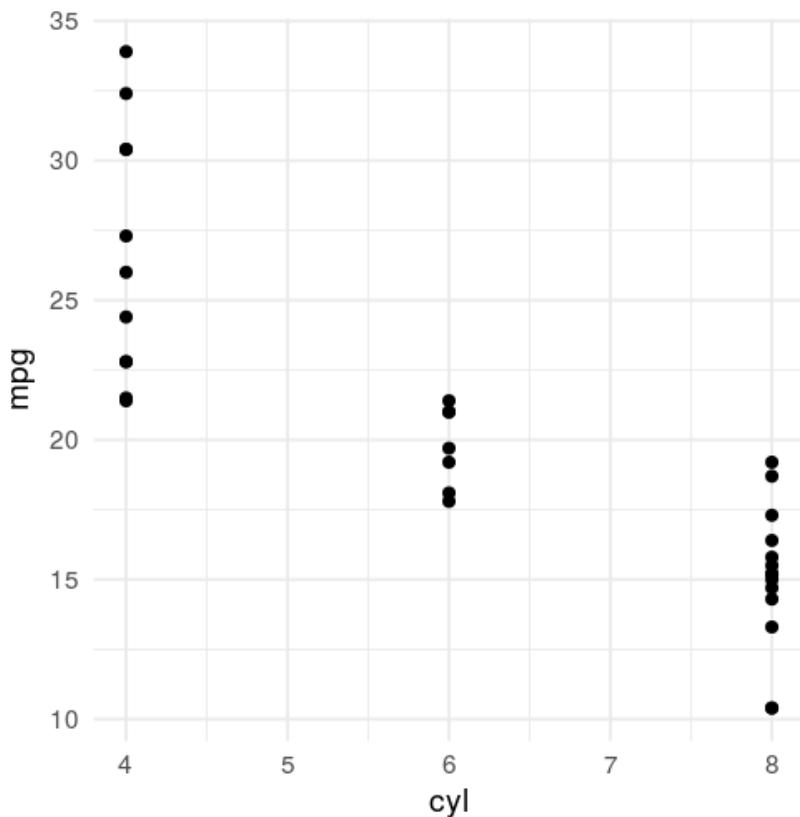
**Figure 9.57. Just the essentials with theme\_classic().**



or remove almost everything with theme\_minimal() (Figure 9.58.).

```
ggplot(mtcars, aes(cyl, mpg)) +  
  geom_point() +  
  theme_minimal()
```

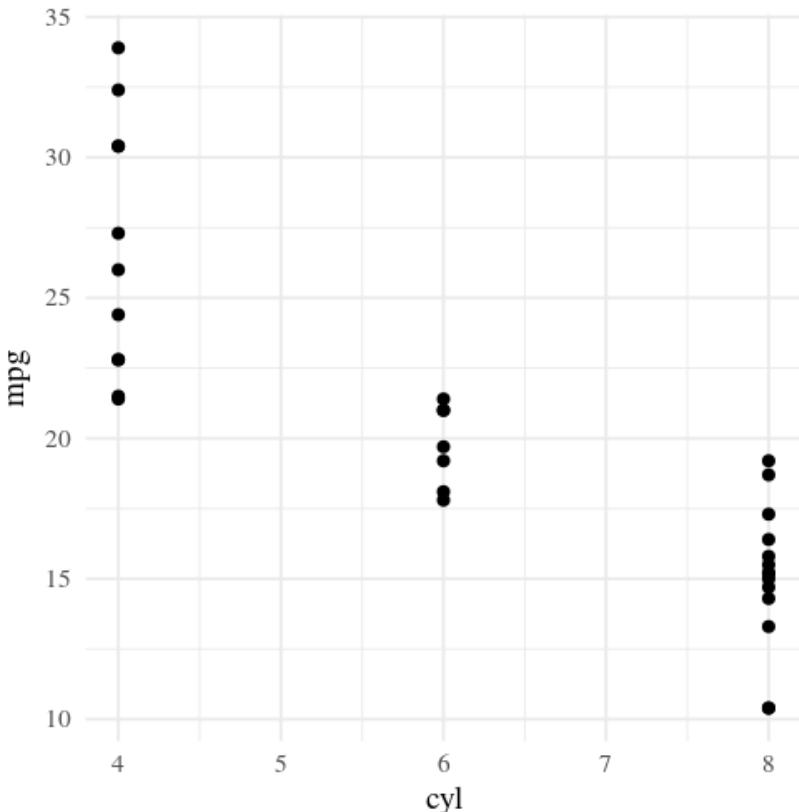
**Figure 9.58.** There is very little apart from the data here.



Many packages exist which contain additional `theme_` functions to extend the possibilities, and you can always specify features yourself if you also call `theme()`, though the syntax there may require you to read through `?theme()` first. For example, we can change the font family to "serif", resulting in [fig-gg-serif](#)

```
ggplot(mtcars, aes(cyl, mpg)) +
  geom_point() +
  theme_minimal() +
  theme(text = element_text(family = "serif"))
```

**Figure 9.59. Changing the text family is achieved through arguments to theme().**



### 9.3 Plots as Objects

One thing to keep in mind is that `ggplot()` returns (invisibly) an object, so you can store a plot just as you would a value. You can then modify it (with some limitations) or add to it. This becomes very handy when you have different features that you want to add to a basic common plot. If we start with an empty plot (with aesthetics/styling and facetting defined) and we save that to the variable `p`

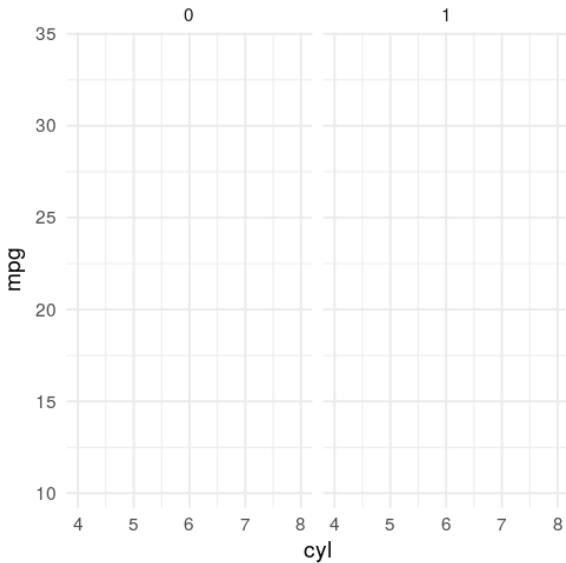
```
p <- ggplot(mtcars, aes(cyl, mpg, col = factor(cyl))) +
  theme_minimal() +
  facet_wrap(~ am)
```



Note that saving our plot to `p` doesn't output anything to the Plot pane since assignment doesn't return anything (visibly). The plot is only generated once you `print()` it, which by default occurs when evaluating a `ggplot()` call on its own. If we try to `print p` (by evaluating it), we get the plot of Figure 9.60.

```
p
```

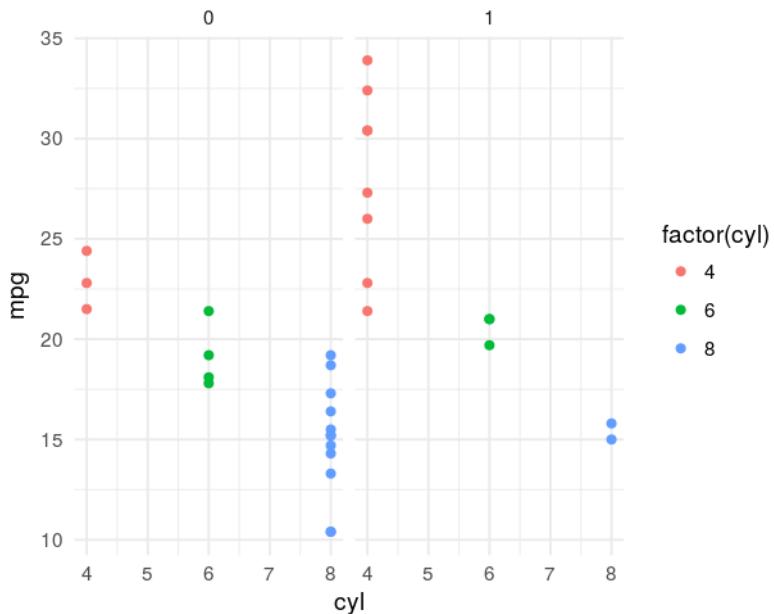
Figure 9.60. A plot produced from the saved variable p.



We can add to p using + as if we had just written the above code, say to add points as in Figure 9.61.

```
p + geom_point()
```

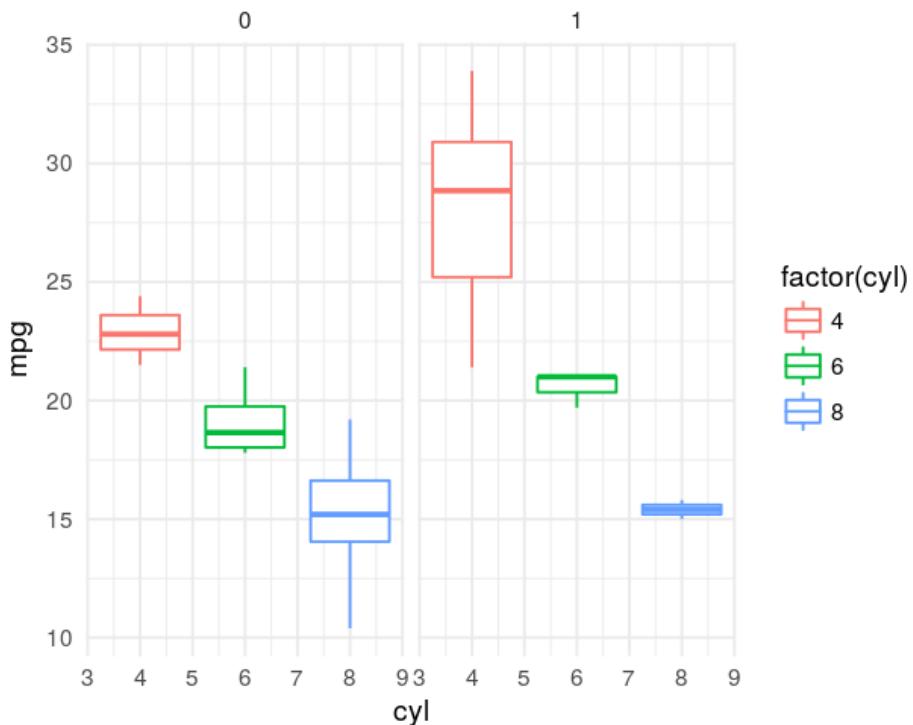
**Figure 9.61. Adding a `geom_point()` call to a variable containing a `ggplot2` plot object.**



or a box and whiskers plot (Figure 9.62.)

```
p + geom_boxplot()
```

**Figure 9.62. Adding a `geom_boxplot()` call to a variable containing a `ggplot2` plot object.**



**ggplot2** isn't a particularly new extension to R; it has a history stretching back more than a decade. Plotting has been available in R (and its predecessors) for at least a decade longer than that, and hasn't changed very much in that time. There is an alternative way to create graphics in R (the base graphics method) and this is presented in [APP-base-graphics](#).

My preference is absolutely towards `ggplot2`; once you're familiar with the syntax you'll be able to throw together a basic inspection of your data in mere seconds. From there the customizable options can take you all the way to a publication-quality graphic. We'll need to know how to keep them of course.

## 9.4 Saving plots

Whichever plotting route you decide to take, at some point you'll most likely want to save some plots for later use. When using RStudio, a handy 'Save as Image' button is available in the 'Export' dropdown at the top of the Plot pane. This provides some control over the file format and size, but it's not a reproducible step towards generating figures from an analysis.

If you're using `ggplot2`, the function `ggsave()` is very convenient, providing scripted

control over the filetype and size of the resulting image, with some clever processing of the input filename to determine exactly how to achieve this; if your chosen filename ends in `.jpg` or `.jpeg` then the corresponding underlying function `jpeg()` will be used to save the plot. Similarly, if your chosen filename ends with `.png` then the underlying function `png()` will be used. If you've saved your plot as an object you can specify this as the `plot` argument explicitly, i.e.

```
p <- ggplot(mtcars, aes(cyl, mpg)) + geom_point()
ggsave(plot = p) ①
```

- ① Explicitly saving the plot stored in the variable `p`, whether or not `p` has been printed on the screen yet.

but if you leave this out then `ggsave()` will assume you mean the last `ggplot2` plot you created (even if you have produced a `base` plot since, and even if you haven't printed the `ggplot2` plot to the screen yet)

```
p <- ggplot(mtcars, aes(cyl, mpg)) + geom_point()
ggsave() ①
```

- ① Implicitly saving the last `ggplot2` plot generated.

Note that for the implicit `ggsave()`, the plot `p` doesn't need to have been printed to the screen at this point; the internal components of `ggplot()` keep a running record of which plot was last generated.<sup>73</sup>

## 9.5 Try It Yourself

 Let's bring together some of the things we've learned in the last few chapters.

## 9.6 Summary

In this chapter we've seen various ways to turn data into a graphical representation. We've seen that there are many ways to do this, and you should now be able to convert some of your own data into a plot and save it. We've introduced the `ggplot2` framework implementing the 'grammer of graphics' and its predecessor `base` graphics. You can now create a reproducible link between your source data and a produced graphic, and can regenerate this if your data changes.

You've learned that:

- Data may need restructuring in order to be visualised
- The `ggplot2` package provides an extensive framework for plotting data
- The features of the 'grammer of graphics' are data, geometries, aesthetic mappings, and scales
- Points, lines, etc... are added using geometries, via `geom_` functions
- When aesthetics are provided in the `ggplot()` part of a call, they are inherited by geometries

<sup>73</sup> You can extract this yourself with `last_plot()`.

- `ggplot2` plots can be stored in variables and added to or saved

New terms you've learned:

#### ***visualisation***

visual representation of some data, most commonly involving two axes.

#### ***aesthetics***

particular aspects of a visualisation; the colour, shape, size, etc... which helps include more dimensions to a visualisation.

#### ***geometry***

the structure in which data is added to a visualisation; lines, points, areas, etc... Multiple geometries can be added to a visualisation when it helps tell the story behind the data.

#### ***grammar of graphics***

Leland Wilkinson's structured system for designing visualisations, involving geometries, aesthetic mappings, scales, and the way in which these each connect to data.

#### ***facet***

one 'side' to the data; when we visualise data with multiple facets we can physically separate them to better highlight similarities within a facet.

#### ***theme***

the overall 'styling' of a visualisation. This is a collection of stylings for the overall look and feel of a visualisation.

Things to remember:

- R plots follow the 'pen and paper' model; once something is added to a plot it can't be removed, and the order in which elements are added determines which will be on top of which.
- Most options are flexible and can be specified, but often these require a bit of searching to find out exactly how. Don't feel bad about this; A question on rotating the x-axis labels is one of the most commonly viewed in the R tag on Stack Overflow. It's my most viewed question too.
- Factors are evenly spaced when plotted, and this is expected behaviour. If you want them spaced unevenly you can use the `scale_` to do this yourself.
- Not everyone can see the colours in every plot the same way; some people might be viewing on a black-and-white printout, some may have vision issues which make distinguishing certain colours difficult. Take care with your visualisation choices to not exclude anyone.

You should now have enough tools to build some visualisations of your data. With a

little more care these can be made publication-quality and included in your reports, articles, blogposts, etc... There are many more extensions to visualisations as this is a popular branch of research, including interactive graphics. For now, we'll move on to how you can put all you've learned so far into a structured form; your own R package.

# 10

## *I Want To Do More With My Data (Extensions)*

**This chapter briefly covers extension topics:**

- Writing your own packages
- Analysing your code (benchmarking/profiling)
- Making code more efficient
- Where to next?
- Communicating your work

If all has gone well, you now have all the tools you need to take command of your data and extract insights from it in a reproducible, readable, and understandable way. That's not the end of the line for learning more about R though; as you gain more experience you'll find certain parts that you wish were better. Maybe your code runs too slowly to be useful, maybe you need more rigor in the functions you've written, maybe you've made something awesome and want to share it with the world. In this chapter we'll cover how to do these should you decide you want to.

There's a lot to learn to fully teach each of these, so I'll cover the basics and hopefully give you enough of a head start to track down the remaining pieces of your particular puzzle.

### **10.1 Writing Your Own Packages**

Whether you have just a single function or a whole directory full of files containing functions, it's often a good idea to solidify these into a package. Not only does this allow some formal testing of your function(s), it provides a very good mechanism for documentation, and at the end of it you have something you can more easily share with others.

I write many of my analyses as packages, even very small ones. This provides me with a way to include the data, perform testing on the functions I create, and wraps it all up in a neat, tidy framework which can be shared or stored.

At its simplest, a minimal R package directory (named as the name of your package) will contain

1. a `DESCRIPTION` file
2. a `NAMESPACE` file
3. a subdirectory named `R`

and that's it.

### 10.1.1 Creating a Minimal Package

The easiest way to achieve this is using the `devtools` package and its function `create()`, passing an (empty) package directory location in as a string. This performs several steps for you:

1. creates a `DESCRIPTION` file with valid defaults
2. creates a `NAMESPACE` file with a catch-all default
3. adds an empty `R` subdirectory
4. creates a 'Project' file `yourPackage.Rproj` (recall [RStudio Projects](#))
5. creates a `.gitignore` file already set up to ignore certain files
6. creates a `.Rbuildignore` already set up to ignore project files.

This takes care of most of the work needed to get a package working. If you open the `yourPackage.Rproj` file (which `create()` just created) in RStudio (e.g. File → Open Project) you'll notice a new Buildtab in one of the panes. This has a button Build & Reload (keyboard shortcut: `Ctrl + Shift + B`) which will trigger the build process and create your package from the above files.

If this goes well, you'll see something like

```
No man pages found in package 'yourPackage'
* installing to library /home/user/R/x86_64-pc-linux-gnu-library/3.4'
* installing *source* package 'yourPackage' ...
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (yourPackage)
```

That `* DONE (yourPackage)` is what you see when a package has successfully been installed, and that's the case here. That means you can now run `library(yourPackage)` and it should work.

That's not much use yet of course, because the package doesn't do anything. We need to provide some functions for the package to provide back to us. If we now create a file `example_function.R` in the `R` directory and write a function in there, such as

```
makeMeSmile <- function() {
  message("I can develop packages now!")
}
```

then re-build the package, we'll again see the `library(yourPackage)` line in

the `Console`. If we start typing the name of our function, it will hopefully autocomplete for us. Evaluating it with no input (it doesn't use any yet), we should see

```
makeMeSmile()
# I can develop packages now!
```

We can return to our script and improve it; let's make it a bit more personal.

 Add an argument to your function, perhaps one which will take your name as a string. Now update the function body to use this information in the `message`.

An example might be

```
makeMeSmile <- function(me = "I") {
  message(me, " can develop packages now!") }
```

① Providing a default argument is often a good idea.

② `message()` takes any number of arguments as `paste0()`s them together before writing to `Console`.

Save the file, and re-build the package. Now try your function in the console with and without the argument.

```
makeMeSmile("You")
# You can develop packages now!
```

### 10.1.2 Documentation

We should of course add some documentation to this function, both for our own benefit and the benefit of anyone using it (even if that is just ourselves). This will be processed into a help file just like any other function has, so we will be able to access it with `?makeMeSmile`. We return to the script in which the function is defined and add some special code. Another extension package to R is the `roxygen2` framework. This provides a mechanism by which we can add documentation of a function in close proximity to the function, which helps keep the two in mind at the same time. This will of course require the `roxygen2` package, which we will need to install

```
# install.packages("roxygen2")
```

We don't actually need to use any of the `roxygen2` functions ourselves, so there's no need to attach these using `library()`; just having the package installed is enough.

The easiest way to achieve this is with `RStudio`; place your cursor anywhere within your function definition and use `Code → Insert Roxygen Skeleton`. When you do that, possibly after installing the latest `roxygen` package, the following will appear just above your function like magic



```
#' Title      1
#'
#' @param me   2
#'
#' @return     3
#' @export     4
```

```
#'
#' @examples
makeMeSmile <- function(me = "I") {
  message(me, " can develop packages now!")
}
```

- ➊ A title which will appear at the top of the help file for your function.
- ➋ Each argument ('parameter') will need to be detailed. By using the 'Insert Roxygen Skeleton' button this is auto-populated with existing argument names.
- ➌ Describe what object your function will return, if anything.
- ➍ If you want your function to be available to a user after they run `library()` you will need your function to be `exported`, in which case it appears in the `NAMESPACE` file.

These lines all begin with `#` so they're treated as comments by the R interpreter, but when used along with an apostrophe as the next character ('), the `roxygen2` package will treat this as special documentation code. `roxygen2` uses a system of tags, which begin with `@` to build up the help file for your function. By default, the first line of the roxygen block denotes the title to be used in your help file. The next few lines can contain the extended description which will appear just below the title as a Description section.

`@param` is used to describe each parameter and creates the Arguments section in the help file; you'll need one per argument in your function, each starting with `@param`. This should start with the name of your argument (mine here is `me`) then be followed by anything the user will need to know about it (default value, restrictions on which values it can take, etc...). Keep in mind that these won't update if you change the function arguments, but having them in such close proximity will hopefully remind you to keep them in sync.

`@return` describes what the function will return and creates the Value section in the help file. In our case, it's nothing (`NULL`) so we can write that.

`@export` is a flag which indicates that the `NAMESPACE` file should be updated to explicitly include this. Exporting a function means it is explicitly available to the user (will show up in autocompletes and can be used easily). Not including this flag indicates that the function is typically for internal use only (say, is needed by another of your functions, but isn't intended to be used by a user). Not marking a function as exported doesn't mean that a user *can't* use it, it just means they'll need to override the namespace search to do so (using `yourPackage:::internalFunction...` note the three colons).

`@examples` is where you show users how to use your function. These show up at the bottom of the help file and should cover most common uses of arguments and corner cases to watch out for. Too often functions either have too few examples or have examples that don't work, so be kind to your users and show them how anything difficult to use works.

Another tag which is good to add is the `@details` tag. This adds a Details section to the help file, and is where you describe the ins and outs of the function.

An example of how this might look for our function is

```

#' Reassure A User That They Are Doing Well
#
#' Look at that, they're building a documented function already!
#
#' @param me the user to reassure
#
#' @return NULL (invisibly). Used for the side-effect of generating a
#'         \code{message}.
#' @export
#
#' @examples
#' makeMeSmile("You")
makeMeSmile <- function(me = "I") {
  message(me, " can develop packages now!")
}

```

Code can be specified (and accordingly styled) using the `\code{}` syntax.

A short but solid example of this in a real-world package is the `roxygen2` documentation for the `tribble()` function from the `tibble` package (as of version 1.4.1.)

```

#' Row-wise tibble creation ①
#
#' Create [tibble]s using an easier to read row-by-row layout. ②
#' This is useful for small tables of data where readability is
#' important. Please see \link{tibble-package} for a general introduction.
#
#' `frame_data()` is an older name for `tribble()`. It will eventually ③
#' be phased out.
#
#' @param ... Arguments specifying the structure of a `tibble`. ④
#' Variable names should be formulas, and may only appear before the data.
#' @return A [tibble]. ⑤
#' @export ⑥
#' @examples ⑦
#
# tribble(
#   ~colA, ~colB,
#   "a", 1,
#   "b", 2,
#   "c", 3
# )
#
#' # tribble will create a list column if the value in any cell is
#' # not a scalar
#' tribble(
#   ~x, ~y,
#   "a", 1:3,
#   "b", 4:6
# )
tribble <- function(...) { (8)
  data <- extract_frame_data_from_dots(...)
  turn_frame_data_into_tibble(data$frame_names, data$frame_rest)
}

```

}

- ➊ The title which appears at the top of the help file.
- ➋ The Description field.
- ➌ Additional Details field.
- ➍ This function only takes the catch-all ... argument and what happens to those values is documented.
- ➎ The return object is described by its class.
- ➏ This function is *exported* so it is available to the user.
- ➐ Some useful examples of usage help users understand how to use your function.
- ➑ Finally, the short code is provided. In this case, there is more documentation than code. Remember to put sufficient time into your own documentation.

This produces the help file `?tribble` which looks like Figure 10. 1.

**Figure 10.1. The help file tribble produced by the above roxygen2 code.**

```
tribble {tibble}                                     R Documentation

Row-wise tibble creation

Description
Create tibbles using an easier to read row-by-row layout. This is useful for small tables of
data where readability is important.
Please see tibble-package for a general introduction.

Usage
tribble(...)

Arguments
... Arguments specifying the structure of a tibble. Variable names should be formulas,
and may only appear before the data.

Details
frame_data() is an older name for tribble(). It will eventually be phased out.

Value
A tibble.

Examples
tribble(
  ~colA, ~colB,
  "a", 1,
  "b", 2,
  "c", 3
)
# tribble will create a list column if the value in any cell is
# not a scalar
tribble(
  ~x, ~y,
  "a", 1:3,
  "b", 4:6
)
```

---

[Package `tibble` version 1.4.1 [Index](#)]

If we re-build and try `?makeMeSmile` we may receive a discouraging sign

```
No documentation for 'makeMeSmile' in specified packages and libraries:
you could try ??makeMeSmile'
```

For whatever reason, the default for some versions of RStudio is set to not document everything on every build. We can change this in the settings. In Build → Configure Build Tools there is a checkbox for 'Generate documentation with Roxygen'. This should be checked. Furthermore, pressing Configure brings up the 'Roxygen Options' menu, the latter part of which has a section 'Automatically roxygenize when running:' and 'Build & Reload' should be checked. Press Okay back to the editor pane. Now when we re-build we should see

```
Writing makeMeSmile.Rd
```

If you try `?makeMeSmile` now, we should see the help page for our function, as in Figure 10. 2.

**Figure 10.2. The help page for the function we just created.**

The screenshot shows the R help page for the `makeMeSmile` function. At the top, it says "R: Reassure A User That They Are Doing Well" and "Find in Topic". On the right, it says "R Documentation". The main title is "Reassure A User That They Are Doing Well". Below it, under "Description", is the text "Look at that, they're building a documented function already!". Under "Usage", the code `makeMeSmile(me = "I")` is shown. Under "Arguments", the argument `me` is described as "the user to reassure". Under "Value", it says "NULL (invisibly). Used for the side-effect of generating a message.". Under "Examples", the code `makeMeSmile("You")` is shown. At the bottom, it says "[Package `yourPackage` version 0.0.0.9000 [Index](#)]".

This is the core of building a package; write and document functions, then build. Lather, rinse, repeat.

## 10.2 Analysing Your Package

Making a package is one thing, but are you sure it does what you think it does? Does it

do that well? One of the strengths of the community-driven nature of the R language is that extensive testing is often performed on packages before they're adopted by the wider community. This instills a level of trust that the package `calculate_x` will actually calculate `x` with some validity.

There are several ways in which we can make a package reliable and trustworthy, and we'll cover some of those briefly here.

### 10.2.1 Unit Testing

Are you certain that your functions do what you expect them to do? For any input? An old rule in programming is to never trust that the user will always use sensible input.

How would you know that the latest change you made to your function didn't break the way it works? Thankfully people have considered this already and we can enlist another extension package to ensure that our assumptions hold true. The `testthat` package provides a mechanism to add *unit tests* to a package.

#### Unit Test

an evaluation which confirms whether or not a particular assumption made by code is true. This is performed at the smallest level possible (a 'unit' of code); while it may be true that a program as a whole 'runs', unit testing performed at the function level ensures that a given input produces a given output (or error, warning, etc...).

```
# install.packages("testthat")
library(testthat)
```

These tests can be evaluated regularly to check assumptions between what a function produces and what is expected. If a small part of our codebase is a function which adds two numbers

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

then we can test that it produces what we expect with different inputs

```
test_that("normal addition rules apply", {
  expect_equal(add(1L, 1L), 2L)          1
  expect_equal(add(-1, 1), 0)             2 3
  expect_error(add(1, "1"))              3
  expect_error(add(1))                  4 5
  expect_type(add(1, 1), "double")       6
  expect_type(add(1L, 1L), "integer")     6
})
```

- ➊ We provide some context to what we're testing.
- ➋ Expectations take the form `expect_x` and test that `x` is true.
- ➌ We expect that the result of the first argument is equal to the second argument.
- ➍ We can also test that code *doesn't* work when we expect it to fail.
- ➎ We can test the expectation that code should fail when improperly used.

- ⑥ We can test that we receive the *type* of data we expect to.

The result of calls to `test_that()` should be error-free and invisible, so we don't need to worry if we see nothing. We know we've made a mistake when that's not true

```
test_that("adding two differently sized data.frames", {
  expect_silent(add(mtcars, iris))
})
```



**Error: Test failed: 'adding two differently sized data.frames' \* '+' only defined for equally-sized data frames <traceback truncated>**

Unit tests are a great way to really poke at your function and see if it does what you think it does. For example, if the user provides a `data.frame` and a number, what will the `add()` function return? We can test it with an assumption; that it should silently work

```
test_that("adding a data.frame to a number", {
  expect_silent(add(mtcars, 1))
})
```

This returns nothing because the evaluation is correct; `add()` (since it's just `+`) can deal with this scenario just fine

```
head(
  add(mtcars, 1)
)
#>          mpg cyl disp hp drat    wt  qsec vs am gear carb
#> Mazda RX4   22.0   7 161 111 4.90 3.620 17.46  1  2  5   5
#> Mazda RX4 Wag 22.0   7 161 111 4.90 3.875 18.02  1  2  5   5
#> Datsun 710   23.8   5 109  94 4.85 3.320 19.61  2  2  5   2
#> Hornet 4 Drive 22.4   7 259 111 4.08 4.215 20.44  2  1  4   2
#> Hornet Sportabout 19.7   9 361 176 4.15 4.440 18.02  1  1  4   3
#> Valiant      19.1   7 226 106 3.76 4.460 21.22  2  1  4   2
```

Every value is increased by 1, compared to the input

```
head(mtcars)
#>          mpg cyl disp hp drat    wt  qsec vs am gear carb
#> Mazda RX4   21.0   6 160 110 3.90 2.620 16.46  0  1  4   4
#> Mazda RX4 Wag 21.0   6 160 110 3.90 2.875 17.02  0  1  4   4
#> Datsun 710   22.8   4 108  93 3.85 2.320 18.61  1  1  4   1
#> Hornet 4 Drive 21.4   6 258 110 3.08 3.215 19.44  1  0  3   1
#> Hornet Sportabout 18.7   8 360 175 3.15 3.440 17.02  0  0  3   2
#> Valiant      18.1   6 225 105 2.76 3.460 20.22  1  0  3   1
```

What if we use two *compatible* `data.frames`?

```
test_that("adding two data.frames", {
  expect_silent(add(mtcars, mtcars))
})
```

Again, `+` is made to deal with this, so

```
head(
```

```

add(mtcars, mtcars)
)
#>          mpg cyl disp hp drat wt qsec vs am gear carb
#> Mazda RX4     42.0 12 320 220 7.80 5.24 32.92 0 2 8 8
#> Mazda RX4 Wag 42.0 12 320 220 7.80 5.75 34.04 0 2 8 8
#> Datsun 710    45.6  8 216 186 7.70 4.64 37.22 2 2 8 2
#> Hornet 4 Drive 42.8 12 516 220 6.16 6.43 38.88 2 0 6 2
#> Hornet Sportabout 37.4 16 720 350 6.30 6.88 34.04 0 0 6 4
#> Valiant       36.2 12 450 210 5.52 6.92 40.44 2 0 6 2

```

We can include tests in a package, so that they can be regularly evaluated when we make changes. The easiest way to accomplish this is with `devtools::use_testthat()`. A good rule of thumb is any time you discover a *bug* in your code (something happened which you didn't expect; a wrong calculation, a broken assumption, an unexpected result on a rarely used input) you should write a unit-test to check for it. To be truly proactive, one can even write the test (knowing what one expects the output to be given an input, regardless of the implementation) *before* writing the actual function/implementation.

This is an extensive topic of its own, so I'll direct you towards the book (<http://r-pkgs.had.co.nz/tests.html>) by the author of that package (again, Hadley Wickham) for more information.

## 10.2.2 Profiling

Code in a package may be documented and tested, but it can also run slower than a week of Sundays. We've been through best practices for making code clear and efficient (e.g. [Tidy Repetition](#)) but what should you do if you find that your function is just taking too long to evaluate?

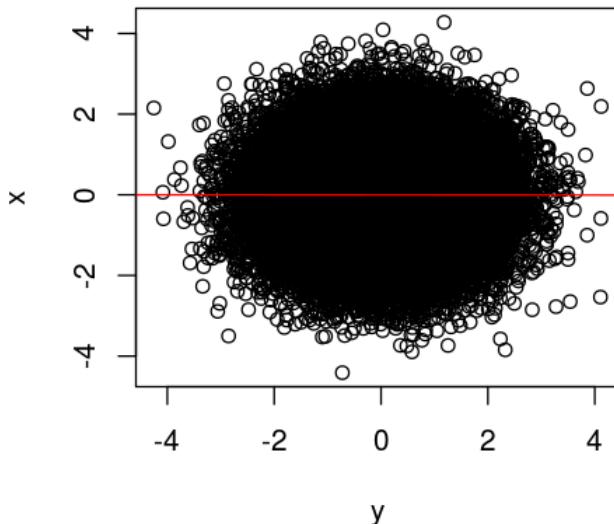
Enter *profiling*; a way to analyse the evaluation of your code at the R level and see which parts are taking the longest to complete. RStudio comes with this functionality built in (thanks to the `profvis` package <https://rstudio.github.io/profvis/>). If we have some code which performs some operations, say, generates some data, calculates a linear model fit between two columns, and plots that line on top of the data (producing Figure 10.3, taken from the help page for `profvis`) and the overall timing of evaluating these isn't impressing us, such as

```

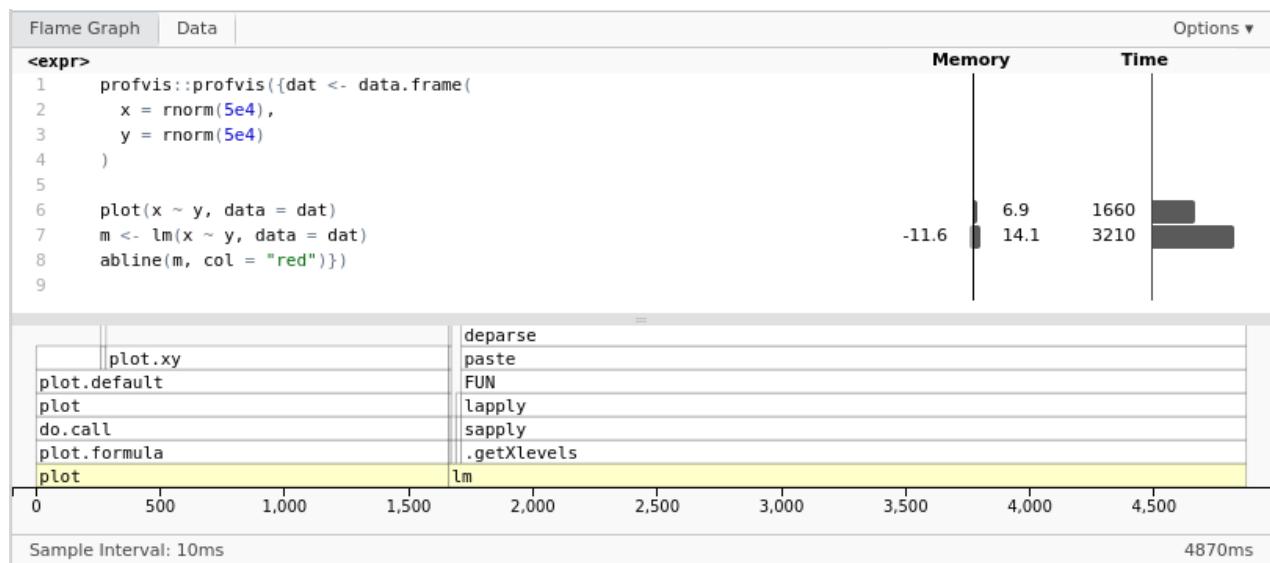
dat <- data.frame(
  x = rnorm(5e4),
  y = rnorm(5e4)
)

plot(x ~ y, data = dat)
m <- lm(x ~ y, data = dat)
abline(m, col = "red")

```

**Figure 10.3.** Example from `?profvis`.

then in this example or our own code, it may not be obvious which line is causing the delay. Is it the data generation? The fitting? The plotting? We can run the profiler over this by highlighting the evaluation lines and using Profile → Profile Selected Lines. This evaluates the code, keeping track of which parts how long to run (including the internal components of each call). We are provided with a timing breakdown in the form of a chart, as shown in Figure 10.4. which compares the memory usage and timing of various components of the call. We can see that the call to `plot()` takes around one third of the total time (1660ms out of the total 4870ms), with the vast majority of the remaining time (3210ms) being due to the `lm()` fitting. The `lm()` step also used the most memory (14.1MB, though it did result in de-allocating 11.6MB also as part of 'R's memory cleanup routines).

**Figure 10.4.** profvis output

While you may have guessed that the fitting would be the slow step, it's not always this obvious which step is taking up all of the processing time.

For more details, see the profvis site <https://rstudio.github.io/profvis/>.

## 10.3 What To Do Next?

Now that you have the basic and intermediate tools at your disposal, it's time to let you loose on your own projects. This is where you'll need to decide for yourself which direction to take, or at least start discovering the more advanced tools you will need to use. Covering every possible data analysis tool would be an insurmountable task; entire books are written on very small aspects of data analysis performed using R. Instead, I'll point you in the direction of some common analysis tasks so you have some vocabulary to begin your search.

### 10.3.1 Regression

One of the most straightforward (in terms of approachability, not interpretation) analyses commonly undertaken is a 'regression'. In terms of R, this involves using functions to determine some relationship between data values. The simplest version of this is a 'linear regression' in which we 'fit' a straight line between two sets of values. This is often referred to as a 'linear model' and the R function which performs this task is named `lm()` as such.

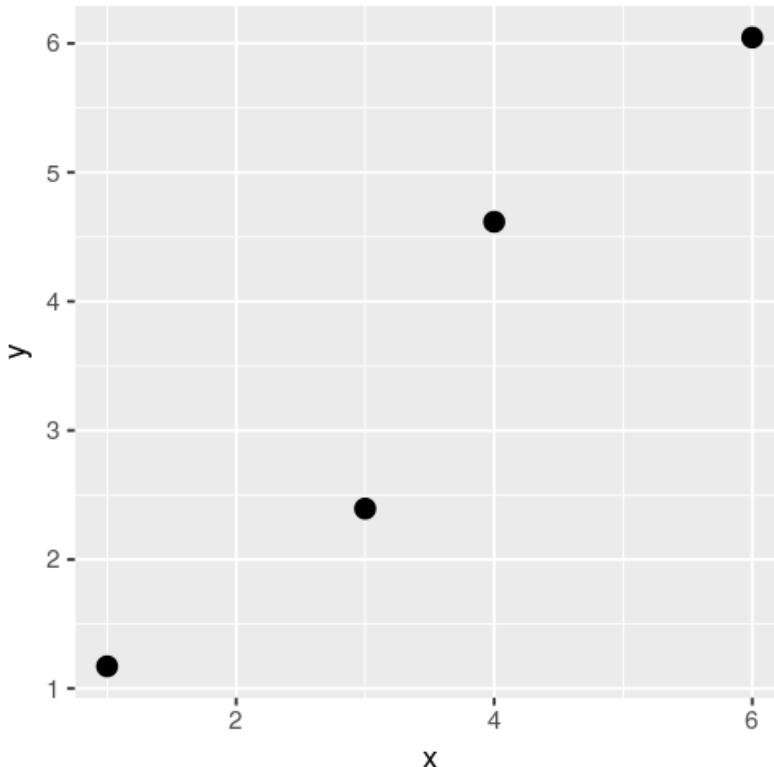
If we have some measured values `y` which were observed at values `x`

```
x <- c(1, 3, 4, 6)
y <- c(1.172, 2.394, 4.617, 6.045)
```

which are plotted in Figure 10.5.

```
# install.packages("ggplot2")
library(ggplot2)
z <- data.frame(x, y)
ggplot(z, aes(x, y)) + geom_point(size = 3) + coord_equal()
```

**Figure 10.5. Values with a linear-like relationship**



then we can fit a linear model to these points using `lm()` which takes a formula (optionally referring to columns in a `data.frame` provided as a `data` argument)

```
linear_model <- lm(y ~ x)
```

The resulting object has its own class `lm` and contains many components which define the model. This class has its own `print` method, so we can produce some output from this object by evaluating it

```
linear_model
#>
#> Call:
#> lm(formula = y ~ x)
#>
```

```
#> Coefficients:
#> (Intercept)           x
#> -0.02215      1.02262
```

We can extract more information by requesting a `summary` of the model

```
summary(linear_model)
#>
#> Call:
#> Lm(formula = y ~ x)
#>
#> Residuals:
#>    1      2      3      4
#> 0.17154 -0.65169 0.54869 -0.06854
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.02215   0.67306 -0.033  0.9767
#> x            1.02262   0.17096  5.982  0.0268 *
#> ---
#> Signif. codes:  0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.6164 on 2 degrees of freedom
#> Multiple R-squared:  0.9471,    Adjusted R-squared:  0.9206
#> F-statistic: 35.78 on 1 and 2 DF,  p-value: 0.02683
```

The slope and intercept are the two coefficients calculated, and these determine the equation describing the line of best fit, a.k.a. linear model, as  $y \sim m*x + c$  for slope  $m$  and intercept  $c$ . More importantly, we can 'predict' what  $y$  values this model would produce at any arbitrary values of  $x$

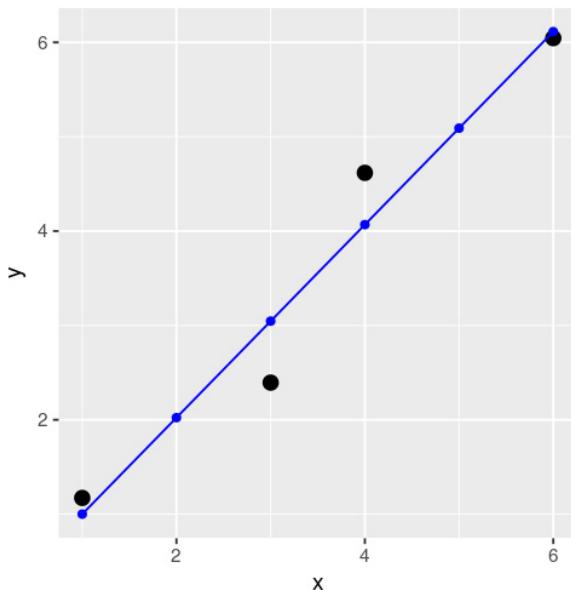
```
prediction_x <- 1:6
prediction_y <- predict(linear_model, newdata = data.frame(x = prediction_x))

predicted <- data.frame(x = prediction_x, y = prediction_y)
predicted
#>   x     y
#> 1 1 1.000462
#> 2 2 2.023077
#> 3 3 3.045692
#> 4 4 4.068308
#> 5 5 5.090923
#> 6 6 6.113538
```

If we plot these predicted  $y$  values at each of these  $x$  values on top of the data points, we see that they follow an exactly straight line; this is the linear model fitted to the data. These are shown in [Fig 10. 6.](#)

```
ggplot(z, aes(x, y)) +
  geom_point(size = 3) +
  geom_point(data = predicted, aes(x, y), col = "blue") +
  geom_line(data = predicted, aes(x, y), col = "blue") +
  coord_equal()
```

**Figure 10.6. Values with a linear-like relationship and line of best fit.**



Not every paired vector of  $x$  and  $y$  values can be well-described by a linear model, but that doesn't stop R or an analyst from doing so. Model selection and interpretation are extensive topics themselves, and I recommend you consult some more specific statistics texts before becoming too attached to any particular model. R places many regression functions at your disposal, including non-linear regression, generalised linear models (GLMs), general additive models (GAMs), and many, many more. R began as a statistical language, so this is certainly a topic in which it is well supported.

### 10.3.2 Clustering

Sometimes the effect you're interested in isn't described by an equation relating some variables, instead you may be interested in 'clusters' within your data; placing your data into two or more groups based on some combination of variables which make them more similar to others within a group than those in other groups. For example, if we have some points which belong to two somewhat distinct groups, even if they appear as one continuous group, such as those in Figure 10.7.

```
# install.packages("dplyr")
library(dplyr)
#
#> Attaching package: 'dplyr'
#> The following object is masked _by_ '.GlobalEnv':
#>
#>     tribble
#> The following object is masked from 'package:testthat':
```

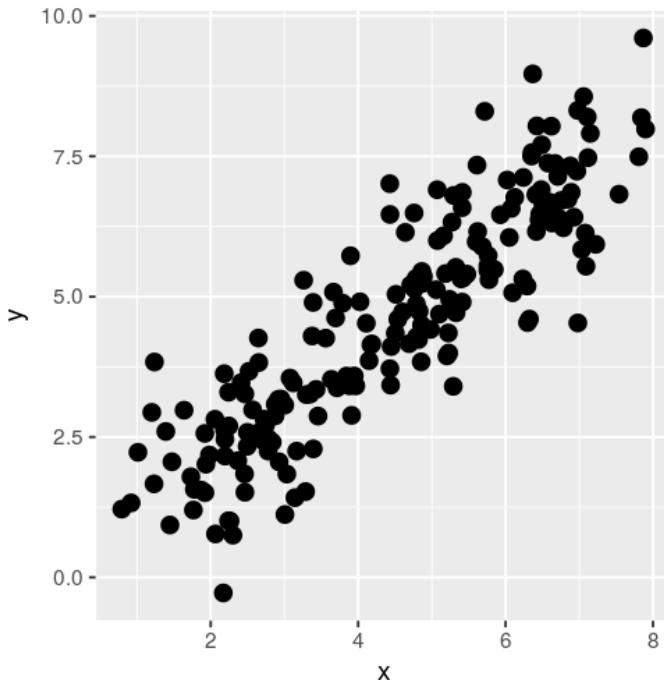
```

#> #>     matches
#> # The following object is masked from 'package:switchr':
#>
#>     location
#> # The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> # The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

sdxy <- 0.9
set.seed(2)
spread_data <- sample_frac( 1
  data.frame(x = c(
    3 + rnorm(100, sd = sdxy),
    6 + rnorm(100, sd = sdxy)
  )) %>% mutate(
    y = x + rnorm(100, sd = sdxy),
    source = rep(c("A", "B"), each = 100)
  ),
  1
)
head(spread_data, 10) 2
#>           x         y source
#> 76  2.183601 3.628281      A
#> 18  3.032226 1.840193      A
#> 75  2.951650 3.177482      A
#> 45  3.560245 4.261439      A
#> 160 6.600770 6.446583      B
#> 28  2.463098 3.267834      A
#> 6   3.119178 3.464633      A
#> 44  4.713193 5.207872      A
#> 148 6.494918 6.748288      B
#> 122 6.124973 6.768940      B
ggplot(spread_data, aes(x, y)) + geom_point(size = 3)

```

**Figure 10.7. Grouped points.** These were produced from two distinct groups but they are spread out enough that they appear like one group.



- ➊ Don't worry too much about the code used to generate these points.
- ➋ Showing the first ten rows of the data, rather than the default of six.

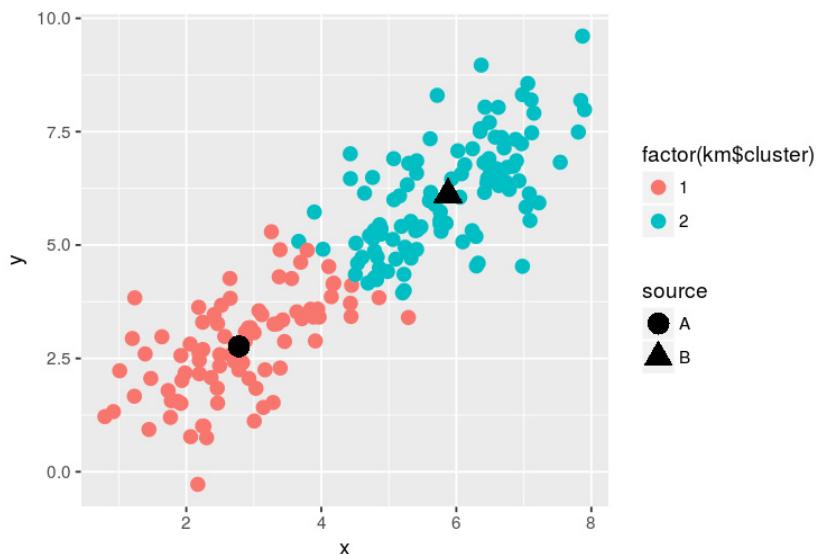
We can use the 'k-means' clustering algorithm to assign each point to one of two clusters (the number of clusters needs to be specified ahead of time)

```
km <- kmeans(spread_data[, c("x", "y")], centers = 2)
str(km)
#> List of 9
#> $ cluster      : Named int [1:200] 1 1 1 1 2 1 1 2 2 2 ...
#> ..- attr(*, "names")= chr [1:200] "76" "18" "75" "45" ...
#> $ centers      : num [1:2, 1:2] 2.78 5.88 2.77 6.12
#> ..- attr(*, "dimnames")=List of 2
#> ...$ : chr [1:2] "1" "2"
#> ...$ : chr [1:2] "x" "y"
#> $ totss        : num 1472
#> $ withinss     : num [1:2] 181 262
#> $ tot.withinss: num 442
#> $ betweenss    : num 1029
#> $ size         : int [1:2] 89 111
#> $ iter         : int 1
#> $ ifault       : int 0
#> - attr(*, "class")= chr "kmeans"
```

The `kmeans()` function performs this task and returns a list containing the clustering information. We can colour our points according to this information at which point the clustering becomes much more apparent, as shown in Figure 10.8.

```
ggplot(spread_data, aes(x, y)) + ❶
  geom_point(size = 3, aes(col = factor(km$cluster))) +
  geom_point(data = data.frame(
    km$centers,
    z = unique(km$cluster)
  ), aes(x, y, shape = c("A", "B")), size = 5) +
  guides(shape = guide_legend(title = "source"))
```

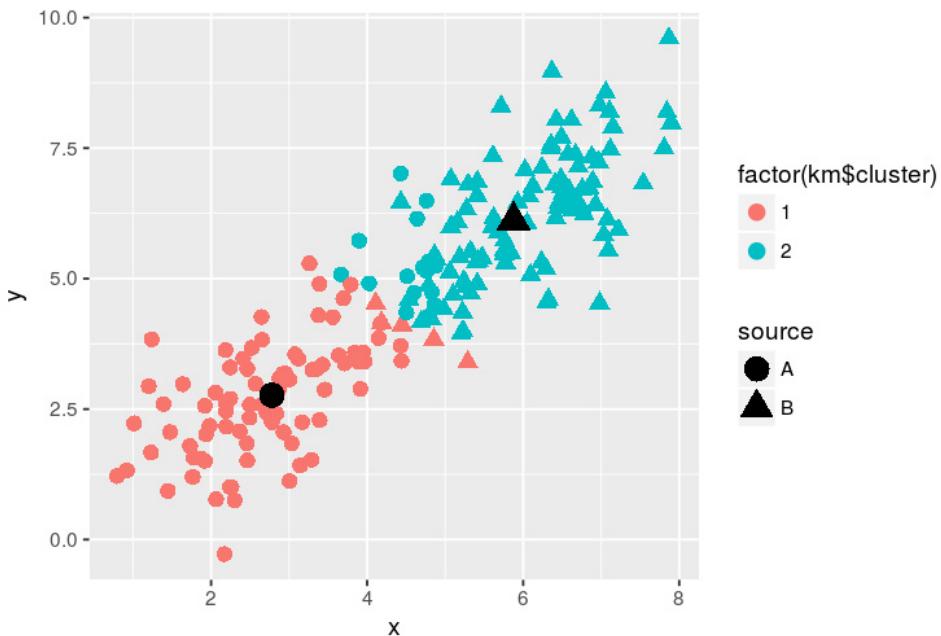
**Figure 10.8. Clustered points. The determined centers of the clusters are shown in black.**



❶ Don't worry too much about the code used to generate the clusters.

Of course, the imaginary line separating these two groups is highly dependent on the exact arrangement of the points themselves; each point is determined to be within a group if it is closer to that group than the other groups, but the tipping point for this condition can be extremely delicate. We can also see that some points have been misidentified (some red triangles, some blue circles) if we also include the 'truth' about which group point belongs to (its 'source') as a shape aesthetic, as shown in Figure 10.9.

```
ggplot(spread_data, aes(x, y)) + ❶
  geom_point(size = 3, aes(col = factor(km$cluster), shape = source)) +
  geom_point(data = data.frame(
    km$centers,
    z = unique(km$cluster)
  ), aes(x, y, shape = c("A", "B")), size = 5)
```

**Figure 10.9.** Notice some misidentified points.

- ➊ Don't worry too much about the code used to generate the clusters.

Many classification and clustering algorithms are available, and again this is a topic worthy of its own book.

### 10.3.3 Working With Maps

We've seen how to place data points on a rectangular map, even join them with lines, but that's hardly the only way we might wish to visualise data. When working with real world data, we might come across some latitude and longitude coordinates. These shouldn't really go on a flat plot with equal x and y scaling; any attempt to do so requires that we consider the 'projection' of a globe onto a flat surface.

#### **Projection**

A transformation of the latitudes and longitudes from the surface of a sphere or an ellipsoid into locations on a flat plane. Since the Earth is not flat, but our paper/screens are, we inevitably require a projection to displace maps, and every projection on to a flat surface creates some distortion.

`ggplot2` is able to deal with this through `coord_*` functions. These modify the plot area such that the gridlines follow a certain projection. For example, if we have map data for the states of the USA, accessible from `ggplot2` via the `maps` package (once you install it, in addition to `ggplot2`) and the `map_data()` function

```
# install.packages("maps") ①
```

- ① Note that we don't need to call `library(maps)`; the internals of `map_data()` just require that the package is installed.

We can then plot it with a flat coordinate projection

```
# install.packages("ggplot2")
library(ggplot2)

states <- map_data(map = "state") ①

head(states)
#>      Long     Lat group order region subregion
#> 1 -87.46201 30.38968    1     1 alabama      <NA>
#> 2 -87.48493 30.37249    1     2 alabama      <NA>
#> 3 -87.52503 30.37249    1     3 alabama      <NA>
#> 4 -87.53076 30.33239    1     4 alabama      <NA>
#> 5 -87.57087 30.32665    1     5 alabama      <NA>
#> 6 -87.58806 30.32665    1     6 alabama      <NA>
```

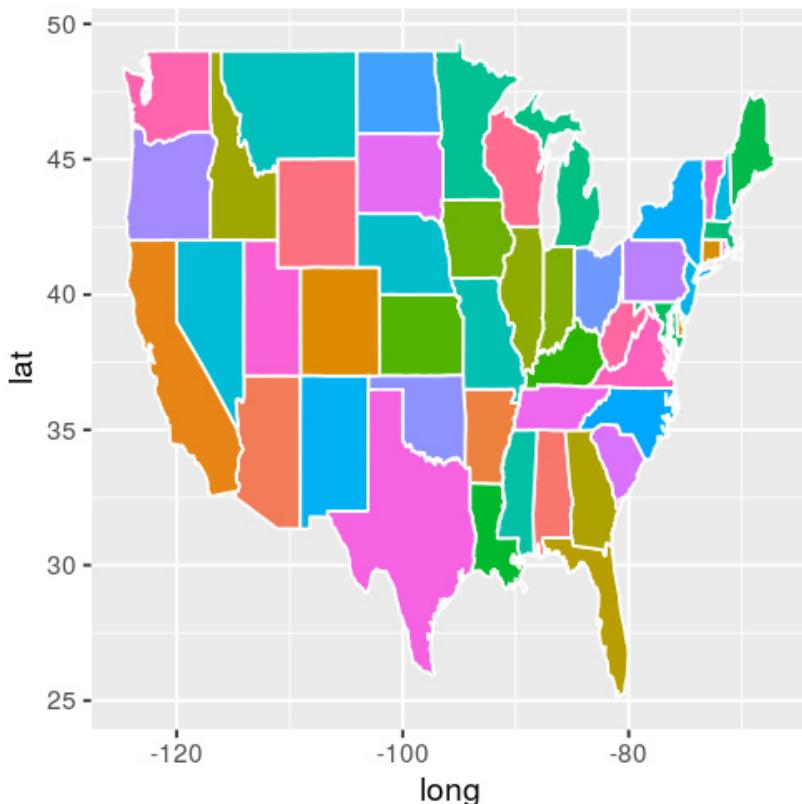
- ① Refer to `?map_data` to see the options available for `map`.

using `ggplot2` as shown in figure 10.10.

```
p <- ggplot(data = states) +
  geom_polygon(aes(x = long, y = lat,
                  fill = region, group = group),
               color = "white") +
  guides(fill=FALSE) ①

p
```

**Figure 10.10. States of the USA. Flat projection.**

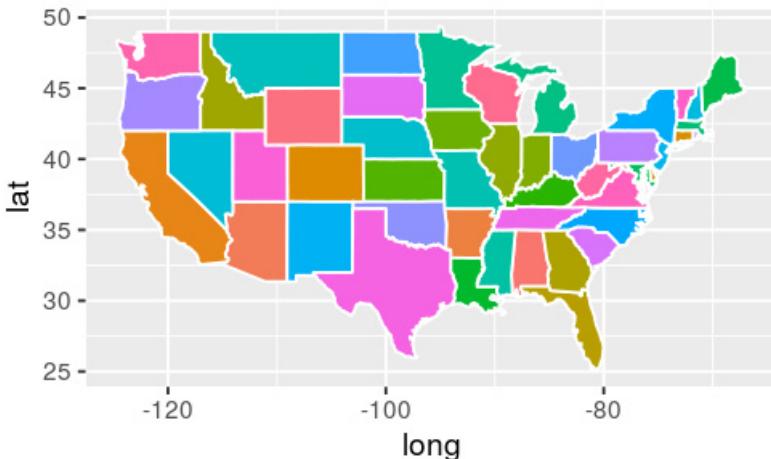


- ➊ We can turn off the fill colour legend for now.

This looks odd because the aspect ratio of the plot isn't correct. We can enforce a projection on this data which constrains this aspect ratio (Figure 10.11.).

```
p + coord_fixed(1.3)
```

**Figure 10.11. Fixing the aspect ratio begins to help but isn't really accurate.**



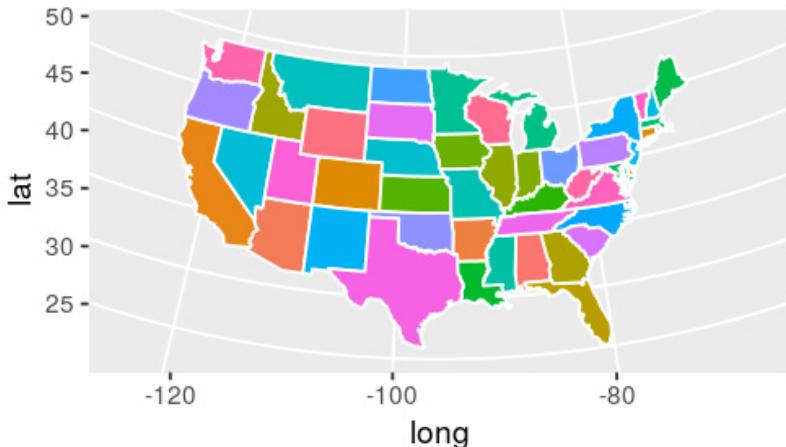
This is roughly correct for this latitude, but to do this correctly we can change to a more suitable projection. In order to do this though, we also require the `mapproj` package to be installed

```
# install.packages("mapproj")
```

(we don't need to use `library()`; `ggplot2` just needs it installed) which allows `ggplot2` to use the `coord_map()` function, resulting in Figure 10.12..

```
p + coord_map(projection = "albers", lat0 = 30, lat1 = 40)
```

**Figure 10.12. A proper projection faithfully represents the map.**



We can of course use any of several other projections available. See `?coord_map` for a list and brief descriptions.

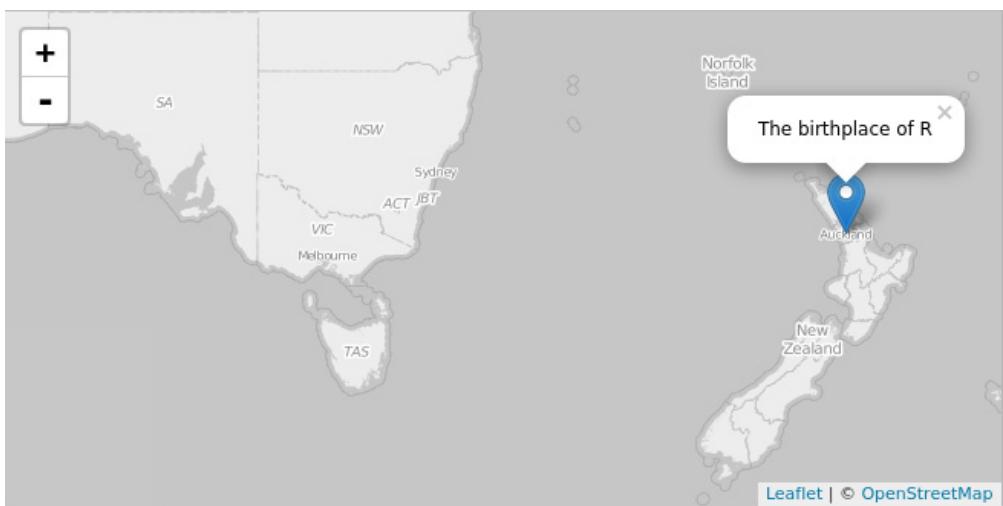
We can also leverage external map providers using the `leaflet` package, producing Figure 10.13.

```
# install.packages("LeafLet")
library(leaflet)

leaflet() %>%
  addProviderTiles("OpenStreetMap.BlackAndWhite") %>% ①
  addMarkers(lng = 174.768, lat = -36.852, popup="The birthplace of R") %>%
  setView(155, -38, zoom = 4)
```

- ① Add OpenStreetMap black and white map tiles. Many providers are available with different map tiles.

**Figure 10.13. A map plot showing the birthplace of R.**



R has made great headway towards replacing more traditional geographic information systems (GIS) and working with spatial data is fast becoming easier with new packages such as `sf`.

#### 10.3.4 Interacting With APIs

If you or I want some information presented to us on a website we click on a link and our computer/phone does the rest for us. It communicates with the hosting server, asks for a page, and presents what it receives in a human-readable way. Computers are happy to jump in a step earlier than that and just deal with the raw response. When the hosting server is configured just right, it provides a programmatic way to interact with it, in the form of an API.

We saw how to scrape Wikipedia in [Scraping Data](#) but we can get a wider view of the entire site by communicating with their API, allowing us to search all of their data rather than starting with a known page. We can request data from servers in general using the `httr` package's function `GET` which performs the tasks that your browser does

when it connects to a server/website.

A simple example of using an API is the wikipedia opensearch facility, a `httr` call to which looks like

```
# install.packages("httr")
library(httr)
wiki_R <- 
content(GET("https://en.wikipedia.org/w/api.php?action=opensearch&search=R&limit=5"))
)
```

This is made up of the API server address (<https://en.wikipedia.org/w/api.php>), and parameters to send as a request. These begin with `?`  to identify them, then they are just name/value pairs; `action=opensearch` (use the 'opensearch' endpoint), `search=R` (search for pages starting with 'R'), and `limit=5` (only return 5 results). There are other parameters which have default values, and these are usually listed in the API documentation, which it is always a good idea to read. What we get back is a *JSON* response,<sup>74</sup> part of which contains the content we are after. It also contains details about the connection (whether or not it was successful, authorized, cookies used, HTML headers found, etc...). We can view this content directly in more recent versions of RStudio as shown in Figure 10. 14.

```
View(wiki_R)
```

**Figure 10.14. Searching the Wikipedia API for pages starting with 'R'.**

Name	Type	Value
wiki_R	list [4]	List of length 4
[[1]]	character [1]	'R'
[[2]]	list [5]	List of length 5
[[1]]	character [1]	'R'
[[2]]	character [1]	'Russia'
[[3]]	character [1]	'Romania'
[[4]]	character [1]	'Record label'
[[5]]	character [1]	'Rail transport'
[[3]]	list [5]	List of length 5
[[1]]	character [1]	'R (named ar/or ) is the 18th letter of the modern English alphabet and the ISO ...'
[[2]]	character [1]	'Russia (: Russian: Россия, tr. Rossija; IPA: [rə'slʲɪjə]), also officially known as the Russian Federation'
[[3]]	character [1]	'Romania ( roh-MAY-nee-ə; Romanian: România [romi'ni.a]) is a sovereign state located in Southeast Europe'
[[4]]	character [1]	'A record label or record company is a brand or trademark associated with the manufacturer of recordings and/or associated with the marketing and distribution of recordings'
[[5]]	character [1]	'Rail transport is a means of transferring of passengers and goods on wheeled vehicles using a rail system'
[[4]]	list [5]	List of length 5
[[1]]	character [1]	'https://en.wikipedia.org/wiki/R'
[[2]]	character [1]	'https://en.wikipedia.org/wiki/Russia'
[[3]]	character [1]	'https://en.wikipedia.org/wiki/Romania'
[[4]]	character [1]	'https://en.wikipedia.org/wiki/Record_label'
[[5]]	character [1]	'https://en.wikipedia.org/wiki/Rail_transport'

<sup>74</sup> JavaScript Object Notation

We can extract elements from this as character vectors

```
## URL for the letter R page
wiki_R[[4]][[1]]
#> [1] "https://en.wikipedia.org/wiki/R"

## Description for the Letter R
strwrap(wiki_R[[3]][[1]], 60) ①
#> [1] "R (named ar/or ) is the 18th letter of the modern English"
#> [2] "alphabet and the ISO basic Latin alphabet."
```

① The output is general text, so here I've line-wrapped it at 60 characters with `strwrap()`.

As another example, the Star Wars API (<https://swapi.co>) provides a wealth of information on the Star Wars canon universe. It also allows computers to request this information directly through API requests. We can gather the information stored for the second record in the 'people' endpoint with

```
C3PO_response <- content(GET("https://swapi.co/api/people/2/"))
```

and again view the resulting content

```
View(C3PO_response)
```

the result of which is shown in Figure 10.15.

**Figure 10.15. Retrieving the second 'people' record from the Star Wars API.**

Name	Type	Value
② C3PO_response	list [16]	List of length 16
name	character [1]	'C-3PO'
height	character [1]	'167'
mass	character [1]	'75'
hair_color	character [1]	'n/a'
skin_color	character [1]	'gold'
eye_color	character [1]	'yellow'
birth_year	character [1]	'112BBY'
gender	character [1]	'n/a'
homeworld	character [1]	'https://swapi.co/api/planets/1/'
③ films	list [6]	List of length 6
④ species	list [1]	List of length 1
vehicles	list [0]	List of length 0
starships	list [0]	List of length 0
created	character [1]	'2014-12-10T15:10:51.357000Z'
edited	character [1]	'2014-12-20T21:17:50.309000Z'
url	character [1]	'https://swapi.co/api/people/2/'

Many APIs are available, but it's not uncommon for these to ask that you provide some authorization through their website before using the service.

### 10.3.5 Sharing Your Package

You've made an efficient, tested, working package; super! Are you the only person who will ever want to see what it does or looks like on the inside? Sharing your work is a great way to get involved in the community and get feedback on the approaches you've taken.

There are several ways to share your work, the most common is to host it on a version control system (VCS) hosting site such as GitHub, GitLab, or Bitbucket. This isn't just a convenient way to backup your work, it also allows others (if you choose to allow them) to view your code and perhaps even make suggestions on how you might improve it. If you take this route, it's a great idea to make use of the README functionality; these sites check repositories for a file named `README.*` (where \* stands for many different file extensions, often `.md` for *markdown*) and presents this as a 'front page' to the repository. Here you can show visitors how to install and use your package, provide notes on how you came to write the code, or maybe cite some relevant sources. This is particularly useful if you're sharing your code with your colleagues in your organisation. You can also set your repository to private if you have the option, and many organisations host their own internal instances of these services.

If you feel that your package is particularly useful to others, you can request to have it hosted on CRAN so that it can be installed by others using `install.packages()`, but be aware that the maintainers have strict requirements (for very good reasons) and you will need to bring your package up to their standards to be eligible. These include passing various checks such as those performed when you use the dropdown menu in the `Build` pane to access More → Check Package or `devtools::check()`. These extensive checks need to pass without error, warning, or even notes if you expect your code to receive a smooth welcome into the CRAN fold.

Once your code is available for public consumption, it's a good idea to write up a blog post or similar to show off what you've made. This not only provides benefit to others in that they learn a bit more about your package, but it's also a great way to sharpen your focus on exactly what you want others to think about when they install and use your package. Which functions are they likely to want to use? Is there anything they need to keep in mind? These thoughts may prompt you to iterate your package and perhaps expand some documentation or provide some warnings in the code.

The Twitter R community is generally warmly welcoming and provides some great amplification of interesting work being performed around the globe. If you post about your package, perhaps linking to your hosted repository, make sure to include the `#rstats` hashtag. Blog posts and Tweets are frequently added to the RWeekly (<https://rweekly.org/>) weekly aggregation through the efforts of editors and community submissions.

## 10.4 More Resources

If you have an idea in mind and want to know what's already available in that space, then the CRAN Task Views page (<https://cran.r-project.org/web/views/>) is a good

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/beyond-spreadsheets-with-r>

Licensed to Asif Qamar <[asif@asifqamar.com](mailto:asif@asifqamar.com)>

place to start. These are curated links to packages (with their descriptions) grouped by topic.

Keeping up with new developments in R-related work is made simpler with the help of aggregators like RWeekly (<https://rweekly.org>) and R-bloggers (<https://www.r-bloggers.com/>). Twitter is a great source of discussion, and there are many dedicated Slack groups hosting more focussed discussions.

If you're completely stuck for ideas, then fork/clone someone's R repository and start tinkering. Break it, see if you can fix it. Find something that needs improving. Document something as you learn more about it. rOpenSci (<https://ropensci.org/packages/>) host many scientifically interesting packages and have a fantastic onboarding process which involves lots of code reviews. Have a look through their resources for inspiration and ideas.

## 10.5 Summary

In this chapter we've seen what we can do now that we know enough of the R language to start creating interesting analyses and work with data. You have the basics now and if you're still interested then there's a wide world of extension packages awaiting you.

You've learned how to:

- Create a minimal R package to contain your code
- Document your code reliably with the roxygen2 package
- Perform unit tests
- Profile some of your code to find bottlenecks
- Perform a simple linear model regression
- Create a projected map visualisation
- Interact with an API using the httr package

New terms you've learned:

### ***unit test***

an evaluation which confirms whether or not a particular assumption made by code is true. This is performed at the smallest level possible (a 'unit' of code); while it may be true that a program as a whole 'runs', unit testing performed at the function level ensures that a given input produces a given output (or error, warning, etc...).

### ***profiling***

automatic examination of the runtimes of individual components within a section of code. Useful for identifying bottlenecks.

### ***regression***

fitting a model (linear or otherwise) to data, usually with the hope of better understanding a mechanism or producing predictions.

***clustering***

assigning values to one of several groups based on their similarities across various variables.

***projection***

(mapping) presenting a curved surface on a flat plane, involving some transformation.

You should now have enough tools at your disposal to go forth and successfully work with the **R** language. Thank you for following along all this way. I wish you all the very best in your endeavours. Please feel free to get in touch with me or Tweet (tag me: @carroll\_jono) if you've found this book useful.