YaRrr! The Pirate's Guide to R

Nathaniel D. Phillips

2017-10-29

Contents

1 Preface **7**
- 1.1 Where did this book come from? 9
- 1.2 Who is this book for? 9
- 1.3 Why is R so great? 9
- 1.4 Why R is like a relationship… 10
- 1.5 R resources 11
- 1.6 Who am I? 16
- 1.7 Please Contribute! 16

2 Getting Started **19**
- 2.1 Installing Base-R and RStudio 19
- 2.2 The four RStudio Windows 21
- 2.3 Packages 25
- 2.4 Reading and writing Code 29
- 2.5 Debugging 30

3 Jump In! **33**
- 3.1 Exploring data 33
- 3.2 Descriptive statistics 35
- 3.3 Plotting 35
- 3.4 Hypothesis tests 41
- 3.5 Regression analysis 42
- 3.6 Bayesian Statistics 43
- 3.7 Wasn't that easy?! 43

4 The Basics **45**
- 4.1 The command-line (Console) 45
- 4.2 Writing R scripts in an editor 47
- 4.3 A brief style guide: Commenting and spacing 48
- 4.4 Objects and functions 51
- 4.5 Test your R might! 56

5 Scalars and vectors **59**
- 5.1 Scalars 59
- 5.2 Vectors 60
- 5.3 Generating random data 63
- 5.4 Test your R might! 69

6 Vector functions **71**
- 6.1 Arithmetic operations on vectors 72
- 6.2 Summary statistics 74
- 6.3 Counting statistics 76

6.4 Missing (NA) values 77
6.5 Standardization (z-score) 77
6.6 Test your R Might! 79

7 Indexing Vectors with [ ] 81
7.1 Numerical Indexing 82
7.2 Logical Indexing 83
7.3 Changing values of a vector 88
7.4 Test your R Might!: Movie data 90

8 Matrices and Dataframes 93
8.1 What are matrices and dataframes? 94
8.2 Creating matrices and dataframes 95
8.3 Matrix and dataframe functions 98
8.4 Dataframe column names 100
8.5 Slicing dataframes 103
8.6 Combining slicing with functions 106
8.7 Test your R might! Pirates and superheroes 108

9 Importing, saving and managing data 111
9.1 Workspace management functions 111
9.2 The working directory 112
9.3 Projects in RStudio 112
9.4 The workspace 114
9.5 .RData files 115
9.6 .txt files 118
9.7 Excel, SPSS, and other data files 120
9.8 Additional tips 121
9.9 Test your R Might! 121

10 Advanced dataframe manipulation 123
10.1 `order()`: Sorting data 124
10.2 `merge()`: Combining data 125
10.3 `aggregate()`: Grouped aggregation 126
10.4 `dplyr` 128
10.5 Additional aggregation functions 130
10.6 Test your R might!: Mmmmm…caffeine 132

11 Plotting (I) 135
11.1 Colors 137
11.2 Plotting arguments 142
11.3 Scatterplot: `plot()` 142
11.4 Histogram: `hist()` 145
11.5 Barplot: `barplot()` 148
11.6 `pirateplot()` 151
11.7 Low-level plotting functions 163
11.8 Saving plots to a file with `pdf()`, `jpeg()` and `png()` 178
11.9 Examples 179
11.10Test your R might! Purdy pictures 181

12 Plotting (II) 185
12.1 More colors 185
12.2 Plot Margins 194
12.3 Arranging plots with `par(mfrow)` and `layout()` 195
12.4 Additional plotting parameters 200

13 Hypothesis Tests 205
13.1 A short introduction to hypothesis tests . . . 206
13.2 Hypothesis test objects: `htest` . . . 211
13.3 T-test: `t.test()` . . . 212
13.4 Correlation: `cor.test()` . . . 216
13.5 Chi-square: `chsq.test()` . . . 218
13.6 Test your R might! . . . 221

14 ANOVA 223
14.1 Full-factorial between-subjects ANOVA . . . 225
14.2 4 Steps to conduct an ANOVA . . . 226
14.3 Ex: One-way ANOVA . . . 229
14.4 Ex: Two-way ANOVA . . . 231
14.5 Type I, Type II, and Type III ANOVAs . . . 234
14.6 Getting additional information from ANOVA objects . . . 235
14.7 Repeated measures ANOVA using the lme4 package . . . 236
14.8 Test your R might! . . . 236

15 Regression 237
15.1 The Linear Model . . . 237
15.2 Linear regression with `lm()` . . . 237
15.3 Comparing regression models with `anova()` . . . 244
15.4 Regression on non-Normal data with `glm()` . . . 245
15.5 Logistic regression with `glm(family = "binomial"` . . . 247
15.6 Test your might! A ship auction . . . 251

16 Custom functions 253
16.1 Why would you want to write your own function? . . . 253
16.2 The structure of a custom function . . . 255
16.3 Using if, then statements in functions . . . 257
16.4 A worked example: `plot.advanced()` . . . 259
16.5 Test your R might! . . . 265

17 Loops 267
17.1 What are loops? . . . 268
17.2 Creating multiple plots with a loop . . . 270
17.3 Updating a container object with a loop . . . 272
17.4 Loops over multiple indices with a design matrix . . . 274
17.5 The list object . . . 275
17.6 Test your R might! . . . 277

18 Solutions 279
18.1 Chapter 4: The Basics . . . 279
18.2 Chapter 5: Scalers and vectors . . . 279
18.3 Chapter 6: Vector Functions . . . 281
18.4 Chapter 7: Indexing vectors with [] . . . 283
18.5 Chapter 8: Matrices and Dataframes . . . 285
18.6 Chapter 13: Hypothesis tests . . . 287
18.7 Chapter 14: ANOVA . . . 290
18.8 Chapter 15: Regression . . . 291

Chapter 1

Preface

The purpose of this book is to help you learn R from the ground-up.

1.1 Where did this book come from?

Let me make something very, very clear…

I did not write this book.

This whole story started in the Summer of 2015. I was taking a late night swim on the Bodensee in Konstanz and saw a rusty object sticking out of the water. Upon digging it out, I realized it was an ancient usb-stick with the word YaRrr inscribed on the side. Intrigued, I brought it home and plugged it into my laptop. Inside the stick, I found a single pdf file written entirely in pirate-speak. After watching several pirate movies, I learned enough pirate-speak to begin translating the text to English. Sure enough, the book turned out to be an introduction to R called The Pirate's Guide to R.

This book clearly has both massive historical and pedagogical significance. Most importantly, it turns out that pirates were programming in R well before the earliest known advent of computers. Of slightly less significance is that the book has turned out to be a surprisingly up-to-date and approachable introductory text to R. For both of these reasons, I felt it was my duty to share the book with the world.

If you or spot any typos or errors, or have any recommendations for future versions of the book, please write me at YaRrr.Book@gmail.com or tweet me @YaRrrBook.

1.2 Who is this book for?

While this book was originally written for pirates, I think that anyone who wants to learn R can benefit from this book. If you haven't had an introductory course in statistics, some of the later statistical concepts may be difficult, but I'll try my best to add brief descriptions of new topics when necessary. Likewise, if R is your first programming language, you'll likely find the first few chapters quite challenging as you learn the basics of programming. However, if R is your first programming language, that's totally fine as what you learn here will help you in learning other languages as well (if you choose to). Finally, while the techniques in this book apply to most data analysis problems, because my background is in experimental psychology I will cater the course to solving analysis problems commonly faced in psychological research.

What this book is

This book is meant to introduce you to the basic analytical tools in R, from basic coding and analyses, to data wrangling, plotting, and statistical inference.

What this book is not

This book does not cover any one topic in extensive detail. If you are interested in conducting analyses or creating plots not covered in the book, I'm sure you'll find the answer with a quick Google search!

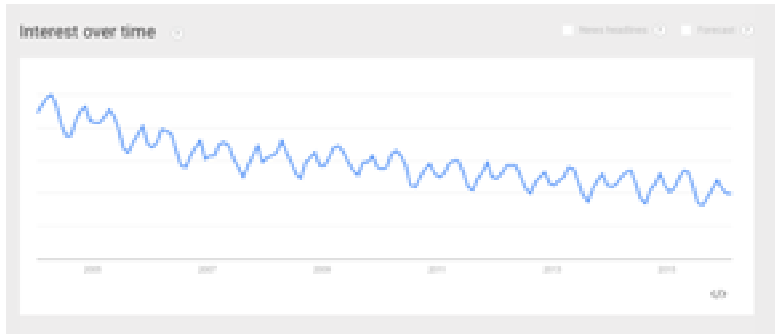
1.3 Why is R so great?

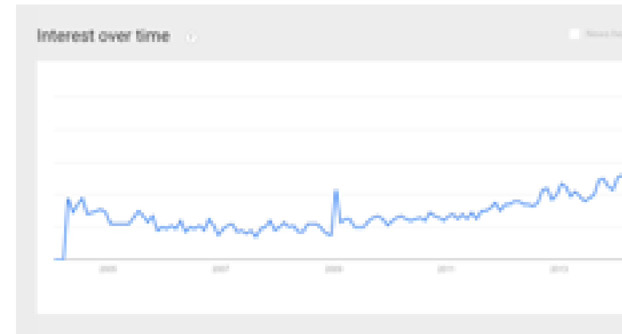
As you've already gotten this book, you probably already have some idea why R is so great. However, in order to help prevent you from giving up the first time you run into a programming wall, let me give you a few more reasons:

1. R is 100% free and as a result, has a huge support community. Unlike SPSS, Matlab, Excel and JMP, R is, and always will be completely free. This doesn't just help your wallet - it means that a huge community of R programmers will constantly develop an distribute new R functionality and packages at a speed that leaves all those other packages in the dust! Unlike Fight Club, the first rule of R is "Do

talk about R!" The size of the R programming community is staggering. If you ever have a question about how to implement something in R, a quick Poogle[1] search will lead you to your answer virtually every single time.

2. R is the present, and future of statistical programming. To illustrate this, look at the following three figures. These are Google trend searches for three terms: R Programming, Matlab, and SPSS. Try and guess which one is which.





3. R is incredibly versatile. You can use R to do everything from calculating simple summary statistics, to performing complex simulations to creating gorgeous plots like the chord diagram on the right. If you can imagine an analytical task, you can almost certainly implement it in R.

4. Using RStudio, a program to help you write R code, You can easily and seamlessly combine R code, analyses, plots, and written text into elegant documents all in one place using Sweave (R and Latex) or RMarkdown. In fact, I translated this entire book (the text, formatting, plots, code...yes, everything) in RStudio using Sweave. With RStudio and Sweave, instead of trying to manage two or three programs, say Excel, Word and (sigh) SPSS, where you find yourself spending half your time copying, pasting and formatting data, images and test, you can do everything in one place so nothing gets misread, mistyped, or forgotten.

```
circlize::chordDiagram(matrix(sample(10),
                              nrow = 2, ncol = 5))
```

5. Analyses conducted in R are transparent, easily shareable, and reproducible. If you ask an SPSS user how they conducted a specific analyses, they will either A) Not remember, B) Try (nervously) to construct an analysis procedure on the spot that makes sense - which may or may not correspond to what they actually did months or years ago, or C) Ask you what you are doing in their house. I used to primarily use SPSS, so I speak from experience on this. If you ask an R user (who uses good programming techniques!) how they conducted an analysis, they should always be able to show you the exact code they used. Of course, this doesn't mean that they used the appropriate analysis or interpreted it correctly, but with all the original code, any problems should be completely transparent!

6. And most importantly of all, R is the programming language of choice for pirates.

1.4 Why R is like a relationship...

Yes, R is very much like a relationship. Like relationships, there are two major truths to R programming:

1. There is nothing more *frustrating* than when your code does *not* work
2. There is nothing more *satisfying* than when your code *does* work!

Anything worth doing, from losing weight to getting a degree, takes time. Learning R is no different. Especially if this is your first experience programming, you are going to experience a *lot* of headaches when

[1] I am in the process of creating Poogle - Google for Pirates. Kickstarter page coming soon...

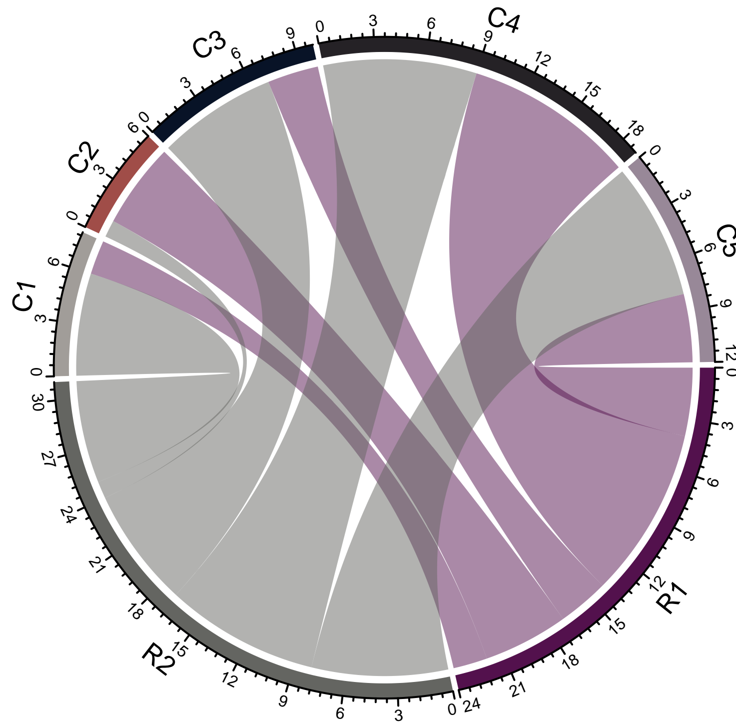


Figure 1.1: A super cool chord diagram from the circlize package

you get started. You will run into error after error and pound your fists against the table screaming: "WHY ISN'T MY CODE WORKING?!?!? There must be something wrong with this stupid software!!!" You will spend hours trying to find a bug in your code, only to find that - frustratingly enough, you had had an extra space or missed a comma somewhere. You'll then wonder why you ever decided to learn R when (::sigh::) SPSS was so "nice and easy."

Fun Fact! SPSS stands for "Shitty Piece of Shitty Shit". True story.

This is perfectly normal! Don't get discouraged and DON'T GO BACK TO SPSS! That would be quitting on exercise altogether because you had a tough workout.

Trust me, as you gain more programming experience, you'll experience fewer and fewer bugs (though they'll never go away completely). Once you get over the initial barriers, you'll find yourself conducting analyses much, much faster than you ever did before.

1.5 R resources

1.5.1 R Cheatsheets

Over the course of this book, you will be learning *lots* of new functions. Wouldn't it be nice if someone created a Cheatsheet / Dictionary of many common R functions? Yes it would, and thankfully several friendly R programmers have done just that. Below is a table of some of them that I recommend. I highly encourage you to print these out and start highlighting functions as you learn them!

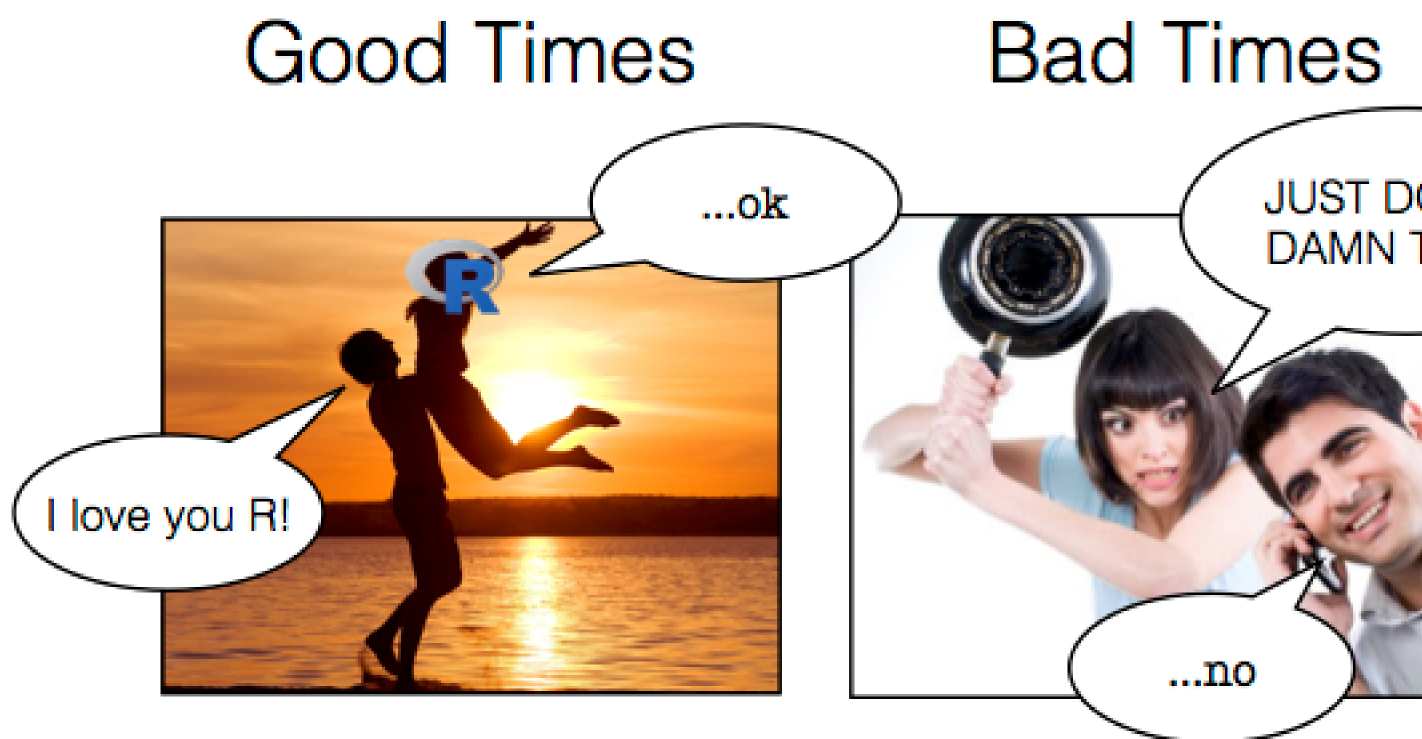


Figure 1.2: Yep, R will become both your best friend and your worst nightmare. The bad times will make the good times oh so much sweeter.

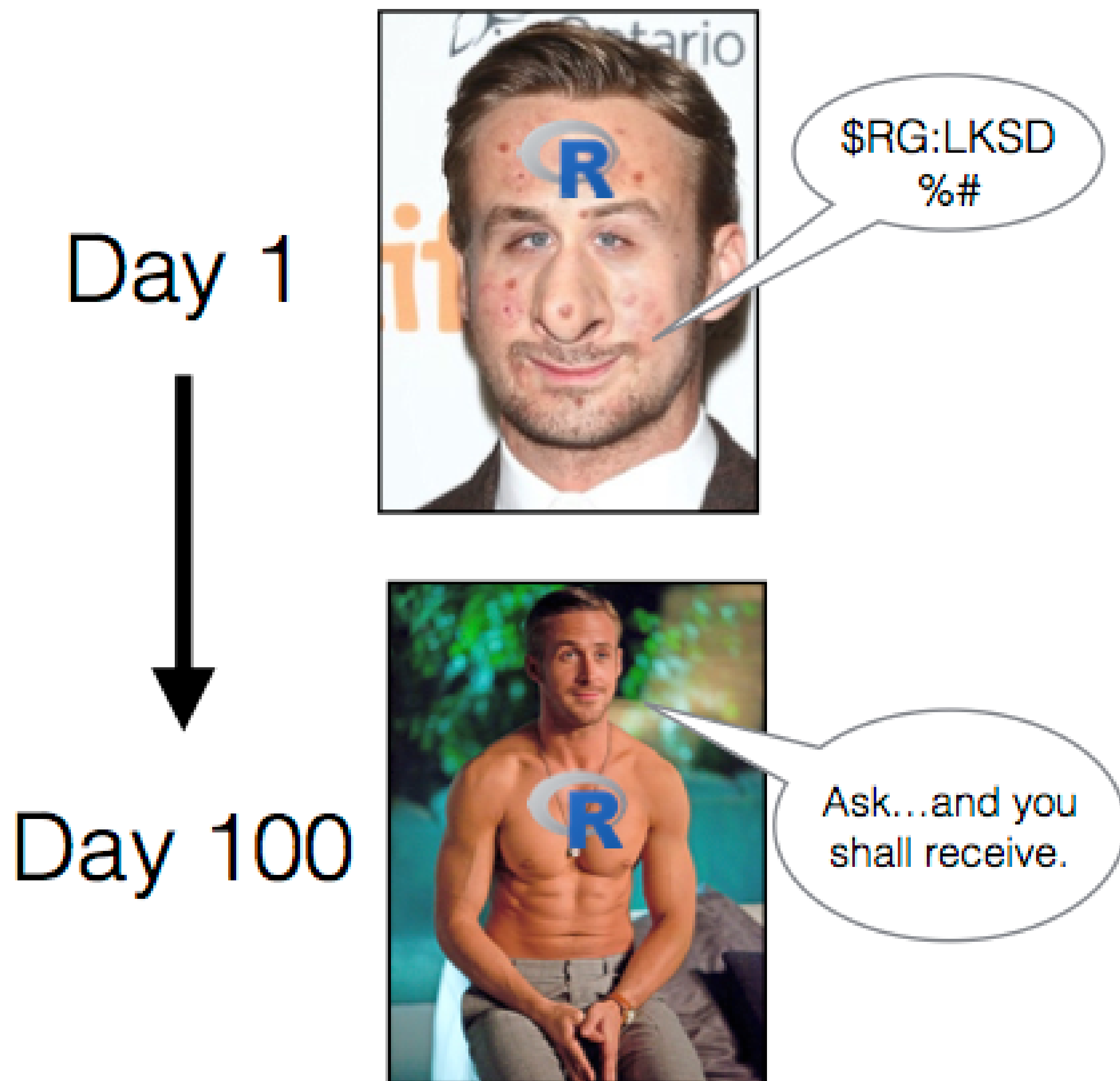


Figure 1.3: When you first meet R, it will look so fugly that you'll wonder if this is all some kind of sick joke. But trust me, once you learn how to talk to it, and clean it up a bit, all your friends will be crazy jealous.

R Reference Card

by Tom Short, EPRI PEAC, tshort@epri-peac.com 2004-11-07
Granted to the public domain. See www.Rpad.org for the source and latest version. Includes material from *R for Beginners* by Emmanuel Paradis (with permission).

Getting help

Most R functions have online documentation.
help(topic) documentation on `topic`
?topic id.
help.search("topic") search the help system
apropos("topic") the names of all objects in the search list matching the regular expression "topic"
help.start() start the HTML version of help
str(a) display the internal *str*ucture of an R object
summary(a) gives a "summary" of `a`, usually a statistical summary but it is *generic* meaning it has different operations for different classes of `a`
ls() show objects in the search path; specify `pat="pat"` to search on a pattern
ls.str() str() for each variable in the search path
dir() show files in the current directory
methods(a) shows S3 methods of `a`
methods(class=class(a)) lists all the methods to handle objects of class a

Input and output

load() load the datasets written with `save`
data(x) loads specified data sets
library(x) load add-on packages
read.table(file) reads a file in table format and creates a data frame from it; the default separator `sep=""` is any whitespace; use `header=TRUE` to read the first line as a header of column names; use `as.is=TRUE` to prevent character vectors from being converted to factors; use `comment.char=""` to prevent `"#"` from being interpreted as a comment; use `skip=n` to skip `n` lines before reading data; see the help for options on row naming, NA treatment, and others

character or factor columns a
field separator; `eol` is the en
missing values; use `col.nam`
get the column headers align
sink(file) output to `file`, until
Most of the I/O functions have a `fil`
ter string naming a file or a connectio
output. Connections can include files
On windows, the file connection c
`"clipboard"`. To read a table copied
`x <- read.delim("clipboard")`
To write a table to the clipboard for E
`write.table(x,"clipboard",sep`
For database interaction, see packa
`ROracle`. See packages `XML`, `hdf5`, `r`

Data creation

c(...) generic function to combi
vector; with `recursive=TRU`
elements into one vector
from:to generates a sequence; ":"
seq(from,to) generates a seque
specifies desired length
seq(along=x) generates `1, 2,`
loops
rep(x,times) replicate `x` `tim`
ement of `x` `each` times;
`rep(c(1,2,3),each=2)` is
data.frame(...) create a d
arguments; `data.frame(v=`
shorter vectors are recycled t
list(...) create a list of
`list(a=c(1,2),b="hi",c=`
array(x,dim=) array with
`dim=c(3,4,2)`; elements of
matrix(x,nrow=,ncol=) matr
factor(x,levels=) encodes a
gl(n,k,length=n*k,labels=
ifying the pattern of their le
the number of replications

Figure 1.4: The R reference card written by Tom Short is absolutely indispensable!

CheatSheet	Link
R Basics by Tom Short	https://cran.r-project.org/doc/contrib/Short-refcard.pdf
R Basics by Mhairi McNeill	http://github.com/rstudio/cheatsheets/raw/master/source/pdfs/base-r.pdf
Advanced R by Arianne Colton and Sean Chen	hhttps://www.rstudio.com/wp-content/uploads/2016/02/advancedR.pdf
Plotting	https://www.rstudio.com/wp-content/uploads/2016/10/how-big-is-your-graph.pdf

1.5.2 Getting R help and inspiration online

Here are some great resources for R help and inspiration:

Site	Description
www.google.com	If you haven't heard of it, Google is this amazing site that gives you access to all R knowledge that has ever existed. Just ask it an R question and 99.9% of the time it will give you the answer!
www.r-bloggers.com	R bloggers is my go-to place to discover the latest and greatest with R.
blog.revolutionanalytics.com	Revolution analytics always has great R related material.
www.kaggle.com	Kaggle is a really cool website that posts data analysis challenges that anyone can try to solve. It also contains a wide range of real-world datasets and tutorials.

1.5.3 Other R books

There are many, many excellent (non-pirate) books on R, some of which are available online for free. Here are some that I highly recommend:

Book	Description
R for Data Science by Garrett Grolemund and Hadley Wickham	The best book to learn the latest tools for elegantly doing data science.
The R Book by Michael Crawley	As close to an R bible as you can get.

Book	Description
Advanced R by Hadley Wickham	A truly advanced book for expert R users, especially those with a programming background. Hadley Wickham is *the* R guru.
Discovering Statistics with R by Field, Miles and Field	A classic text focusing on the theory and practice of statistical analysis with R
Applied Predictive Modeling by Kuhn and Johnson	A great text specializing in statistical learning aka predictive modeling aka machine learning with R.

1.6 Who am I?

My name is Nathaniel – not Nathan…not Nate…and *definitely* not Nat. I am a psychologist with a background in statistics and judgment and decision making. You can find my R (and non-R) related musings at http://ndphillips.github.io

1.6.1 Acknowledgements

I am deeply indebted to many people for either directly or indirectly helping me make this book happen. I would especially like to thank Captain Thomas Moore and Captain Wei Linn for my early training in both statistics and R, Captain Hansjoerg Neth for teaching me LaTeX and ultimately inspiring me to write (I mean translate) this book, and Captain Dirk Wulff for teaching me almost everything I know about R. If I hadn't been lucky enough to meet just one of these people, this book would not exist.

1.7 Please Contribute!

I am grateful for comments, questions, bug reports, and requests to future editions of the book! If there's anything you'd like to add or share, please contact me via email at yarrr.book@gmail.com, or if you are familiar with GitHub, post an issue at https://github.com/ndphillips/ThePiratesGuideToR/issues. Contributers will be added to the R pirate

Figure 1.5: Like a pirate, I work best with a mug of beer within arms' reach.

Chapter 2

Getting Started

2.1 Installing Base-R and RStudio

To use R, we'll need to download two software packages: **Base-R**, and **RStudio**. Base-R is the basic software which contains the R programming language. RStudio is software that makes R programming easier. Of course, they are totally free and open source.

2.1.1 Check for version updates

R and RStudio have been around for several years – however, they are *constantly* being updated with new features and bug-fixes. At the time that I am writing this sentence (09:48, Thursday, 23 February, 2017), the latest version of Base-R is 3.3.2 "Sincere Pumpkin Patch" (the versions all have funny names) which was released on 31 October, 2016, and the latest version of RStudio is 1.0.136 released on 21 December, 2016. If you have a (much) older version of R or RStudio currently installed on your computer, then you should update both R and RStudio to the newest version(s) by installing them again from scratch. If you don't, then some of the code and packages in this book might not work.

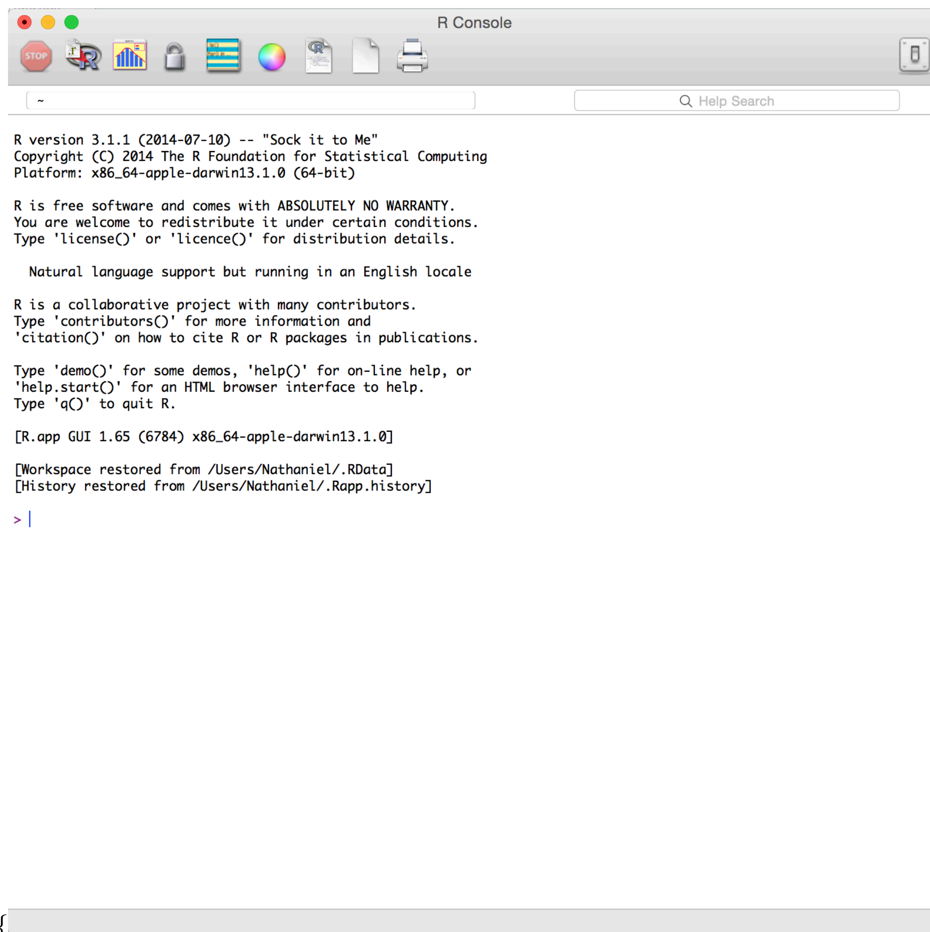
To install Base-R, click on one of the following links and follow the instructions.

Operating System	Link
Windows	http://cran.r-project.org/bin/windows/base/

Operating System	Link
Mac	http://cran.r-project.org/bin/macosx/

Once you've installed base-R on your computer, try opening it. When you do you should see a screen like the one in Figure 2.1.1 (this is the Mac version). As you can see, base R is very much bare-bones software. It's kind of the equivalent of a simple text editor that comes with your computer.

\begin{figure}

{

}

\caption{{Here is how the base R application looks. While you can use the base R application alone, most people I know use RStudio – software that helps you to write and use R code more efficiently!} \end{figure}

While you can do pretty much everything you want within base-R, you'll find that most people these days do their R programming in an application called RStudio. RStudio is a graphical user interface (GUI)-like interface for R that makes programming in R a bit easier. In fact, once you've installed RStudio, you'll likely never need to open the base R application again. To download and install RStudio (around 40mb), go to one of the links above and follow the instructions.

Operating System	Link
All	http://www.rstudio.com/products/rstudio/download/

Let's go ahead and boot up RStudio and see how she looks!

2.2 The four RStudio Windows

When you open RStudio, you'll see the following four windows (also called panes) shown in in Figure 2.1. However, your windows might be in a different order that those in Figure 2.1. If you'd like, you can change the order of the windows under RStudio preferences. You can also change their shape by either clicking the minimize or maximize buttons on the top right of each panel, or by clicking and dragging the middle of the borders of the windows.

Now, let's see what each window does in detail.

2.2.1 Source - Your notepad for code

The source pane is where you create and edit "R Scripts" - your collections of code. Don't worry, R scripts are just text files with the ".R" extension. When you open RStudio, it will automatically start a new Untitled script. Before you start typing in an untitled R script, you should always save the file under a new file name (like, "2015PirateSurvey.R"). That way, if something on your computer crashes while you're working, R will have your code waiting for you when you re-open RStudio.

You'll notice that when you're typing code in a script in the Source panel, R won't actually evaluate the code as you type. To have R actually evaluate your code, you need to first 'send' the code to the Console (we'll talk about this in the next section).

There are many ways to send your code from the Source to the console. The slowest way is to copy and paste. A faster way is to highlight the code you wish to evaluate and clicking on the "Run" button on the top right of the Source. Alternatively, you can use the hot-key "Command + Return" on Mac, or "Control + Enter" on PC to send all highlighted code to the console.

2.2.2 Console: R's Heart

The console is the heart of R. Here is where R actually evaluates code. At the beginning of the console you'll see the character `>`. This is a prompt that tells you that R is ready for new code. You can type code directly into the console after the `>` prompt and get an immediate response. For example, if you type 1+1 into the console and press enter, you'll see that R immediately gives an output of 2.

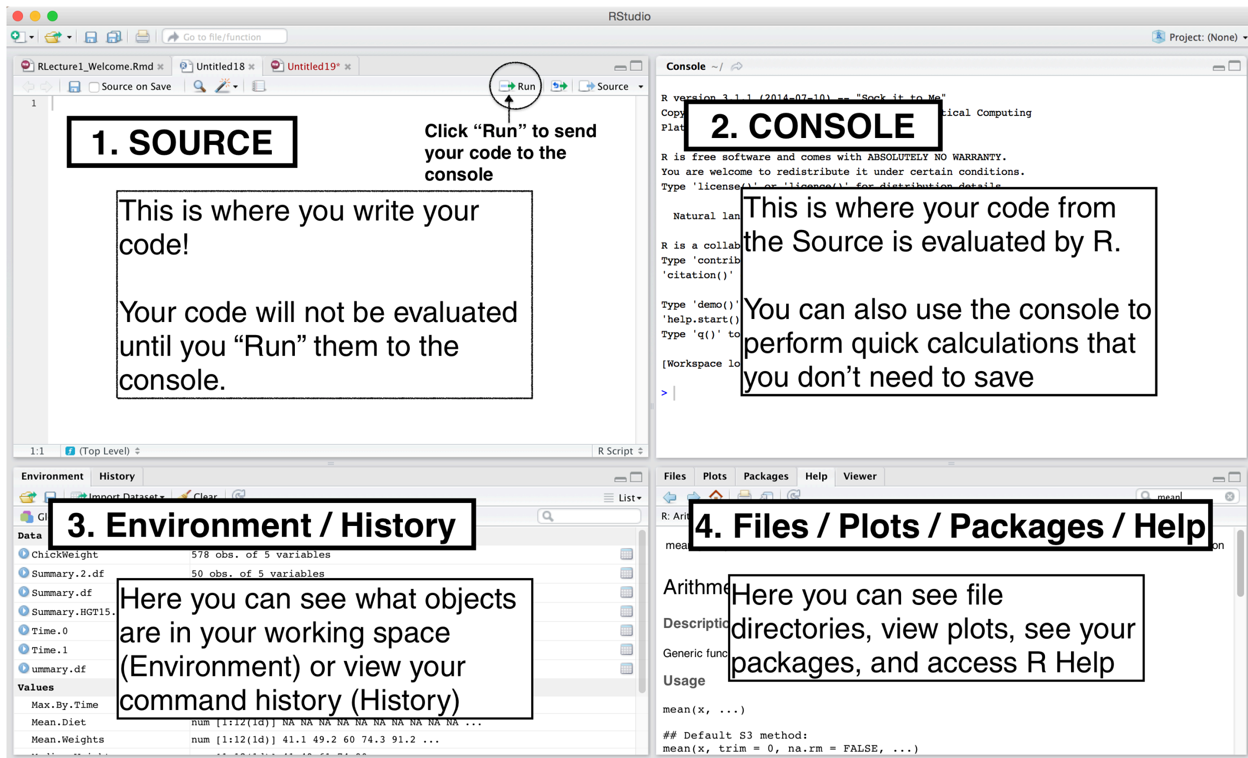


Figure 2.1: The four panes of RStudio.

```
1+1
## [1] 2
```

Try calculating 1+1 by typing the code directly into the console - then press Enter. You should see the result [1] 2. Don't worry about the [1] for now, we'll get to that later. For now, we're happy if we just see the 2. Then, type the same code into the Source, and then send the code to the Console by highlighting the code and clicking the "Run" button on the top right hand corner of the Source window. Alternatively, you can use the hot-key "Command + Return" on Mac or "Control + Enter" on Windows.

Tip: Try to write most of your code in a document in the Source. Only type directly into the Console to de-bug or do quick analyses.

So as you can see, you can execute code either by running it from the Source or by typing it directly into the Console. However, 99% most of the time, you should be using the Source rather than the Console. The reason for this is straightforward: If you type code into the console, it won't be saved (though you can look back on your command History). And if you make a mistake in typing code into the console, you'd have to re-type everything all over again. Instead, it's better to write all your code in the Source. When you are ready to execute some code, you can then send "Run" it to the console.

2.2.3 Environment / History

The Environment tab of this panel shows you the names of all the data objects (like vectors, matrices, and dataframes) that you've defined in your current R session. You can also see information like the number of observations and rows in data objects. The tab also has a few clickable actions like "Import Dataset" which will open a graphical user interface (GUI) for important data into R. However, I almost never look at this menu.

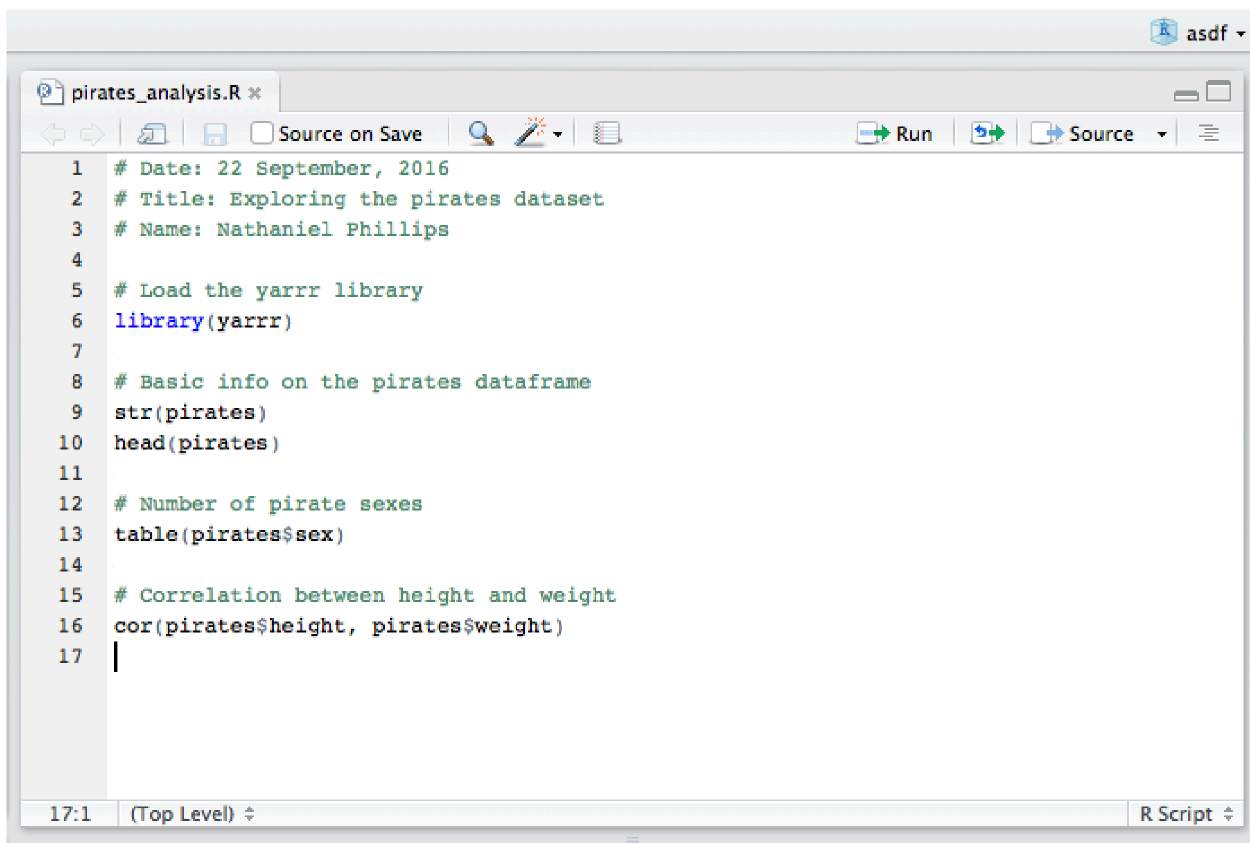


Figure 2.2: The Source contains all of your individual R scripts. The code won't be evaluated until you send it to the Console.

```
Console ~/Desktop/asdf/

R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
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.

> |
```

Figure 2.3: The console the calculation heart of R. All of your code will (eventually) go through here.

Environment | History | Plots | Git | Presentation

Import Dataset ▾ | List ▾

Global Environment ▾

Name	Value
dist.y	int [1:10] 61 44 16 5 9 5 2 1 0 0
f1	chr [1:100] "a" "b" "b" "c" "c" "b" "b" "a" "a" "a" ...
f2	chr [1:100] "a" "a" "b" "c" "a" "b" "b" "a" "a" "a" ...
failure.percent	num [1:4] 0.5 1 0.03 0.97
failures.i	0.97
favpirate	List of 13
file.i	"manchego/brie.csv"
files.to.clean	chr [1:3] "brie/brie.csv" "gruyere/brie.csv" "manche...
fixed.return	num [1:10] 1.05 1.05 1.05 1.05 1.05 1.05 1.05 1.05 1...
fixed.return.prod	131.501257846304
fixed.seq	num [1:100] 1 1.05 1.1 1.16 1.22 ...
full	Formal class BFBayesFactor
g1	num [1:100] 52.2 46.5 50.7 40.9 50.8 ...
g1.c	num [1:100] 52.2 46.5 50.7 40.9 50.9 ...
g1g3.mod	List of 12
g2	num [1:100] 57.4 57.8 54.4 57.7 53.9 ...
g2.c	num [1:100] 47.5 47.8 44.4 47.7 44 ...
g3	num [1:100] 45.2 39.4 35.9 31.1 33 ...
g3.c	num [1:100] 55.4 49.6 46.1 41.3 43.2 ...

Figure 2.4: The environment panel shows you all the objects you have defined in your current workspace. You'll learn more about workspaces in Chapter 7.

The History tab of this panel simply shows you a history of all the code you've previously evaluated in the Console. To be honest, I never look at this. In fact, I didn't realize it was even there until I started writing this tutorial.

As you get more comfortable with R, you might find the Environment / History panel useful. But for now you can just ignore it. If you want to declutter your screen, you can even just minimize the window by clicking the minimize button on the top right of the panel.

2.2.4 Files / Plots / Packages / Help

The Files / Plots / Packages / Help panel shows you lots of helpful information. Let's go through each tab in detail:

1. Files - The files panel gives you access to the file directory on your hard drive. One nice feature of the "Files" panel is that you can use it to set your working directory - once you navigate to a folder you want to read and save files to, click "More" and then "Set As Working Directory." We'll talk about working directories in more detail soon.

2. Plots - The Plots panel (no big surprise), shows all your plots. There are buttons for opening the plot in a separate window and exporting the plot as a pdf or jpeg (though you can also do this with code using the `pdf()` or `jpeg()` functions.)

Let's see how plots are displayed in the Plots panel. Run the code on the right to display a histogram of the weights of chickens stored in the `ChickWeight` dataset. When you do, you should see a plot similar to the one in Figure 2.5 show up in the Plots panel.

```
hist(x = ChickWeight$weight,
     main = "Chicken Weights",
     xlab = "Weight",
     col = "skyblue",
     border = "white")
```

3. Packages - Shows a list of all the R packages installed on your harddrive and indicates whether or not they are currently loaded. Packages that are loaded in the current session are checked while those that are installed but not yet loaded are unchecked. We'll discuss packages in more detail in the next section.

4. Help - Help menu for R functions. You can either type the name of a function in the search window, or use the code `?function.name` to search for a function with the name `function.name`

```
?hist   # How does the histogram function work?
?t.test # What about a t-test?
```

2.3 Packages

When you download and install R for the first time, you are installing the Base R software. Base R will contain most of the functions you'll use on a daily basis like `mean()` and `hist()`. However, only functions written by the original authors of the R language will appear here. If you want to access data and code written by other people, you'll need to install it as a *package*. An R package is simply a bunch of data, from functions, to help menus, to vignettes (examples), stored in one neat package.

A package is like a light bulb. In order to use it, you first need to order it to your house (i.e.; your computer) by *installing* it. Once you've installed a package, you never need to install it again. However, every time you want to actually use the package, you need to turn it on by *loading* it. Here's how to do it.

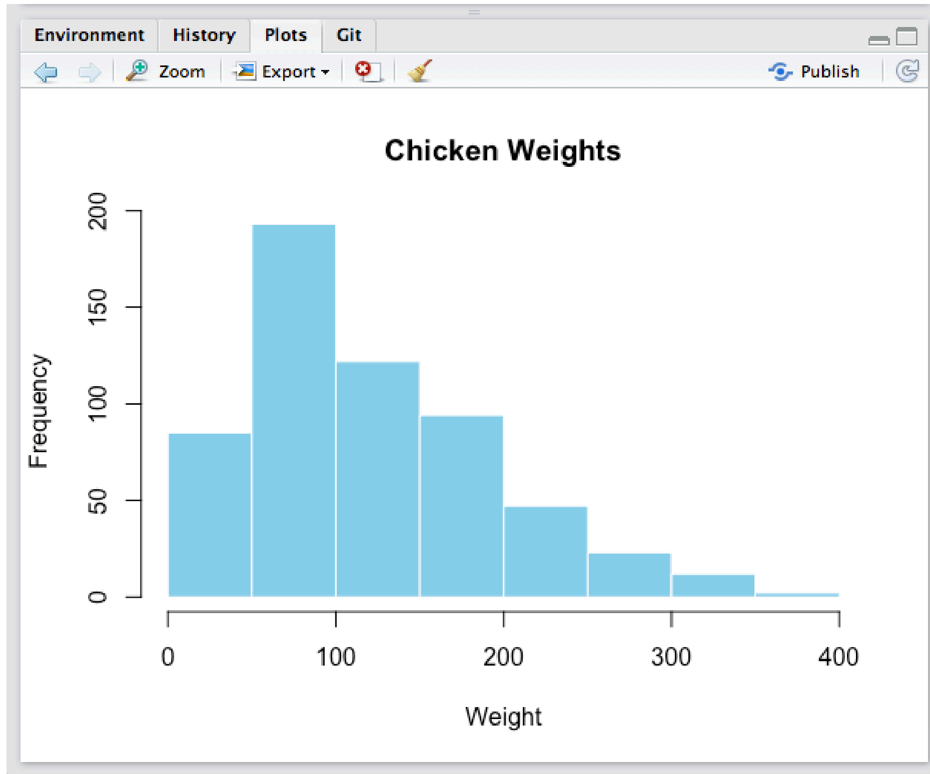


Figure 2.5: The plot panel contains all of your plots, like this histogram of the distribution of chicken weights.

Installing a package

`install.packages('my.package')`

Loading a package

`library('mypackage')`

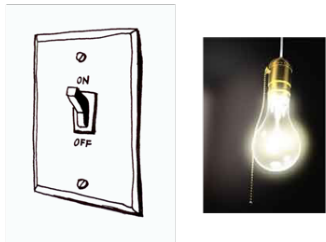


Figure 2.6: An R package is light a lightbulb. First you need to order it with install.packages(). Then, every time you want to use it, you need to turn it on with library()

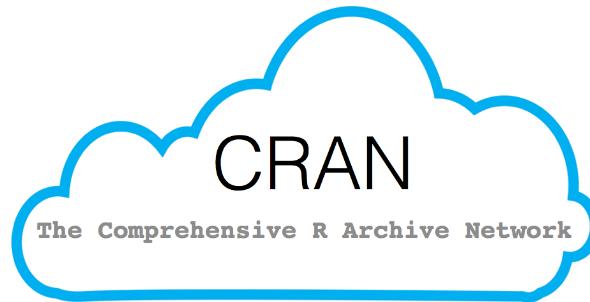


Figure 2.7: CRAN (Comprehensive R Archive Network) is the main source of R packages

```
> install.packages("circlize")
trying URL 'https://cran.rstudio.com/bin/macosx/mavericks/contrib/3.3/circlize_0.3.8.tgz'
Content type 'application/x-gzip' length 3856952 bytes (3.7 MB)
==================================================
downloaded 3.7 MB


The downloaded binary packages are in
	/var/folders/gy/bsyxftvn37q93cm6vt1v56fr0000gp/T//RtmpOzgZ2R/downloaded_packages
>
```

Figure 2.8: When you install a new package, you'll see some random text like this you the download progress. You don't need to memorize this.

2.3.1 Installing a new package

Installing a package simply means downloading the package code onto your personal computer. There are two main ways to install new packages. The first, and most common, method is to download them from the Comprehensive R Archive Network (CRAN). CRAN is the central repository for R packages. To install a new R package from CRAN, you can simply run the code `install.packages("name")`, where "name" is the name of the package. For example, to download the `yarrr` package, which contains several data sets and functions we will use in this book, you should run the following:

```
# Install the yarrr package from CRAN
#   You only need to install a package once!
install.packages("yarrr")
```

When you run `install.packages("name")` R will download the package from CRAN. If everything works, you should see some information about where the package is being downloaded from, in addition to a progress bar.

Like ordering a light bulb, once you've installed a package on your computer you never need to install it again (unless you want to try to install a new version of the package). However, every time you want to use it, you need to turn it on by loading it.

2.3.2 Loading a package

Once you've installed a package, it's on your computer. However, just because it's on your computer doesn't mean R is ready to use it. If you want to use something, like a function or dataset, from a package you *always* need to *load* the package in your R session first. Just like a light bulb, you need to turn it on to use it!

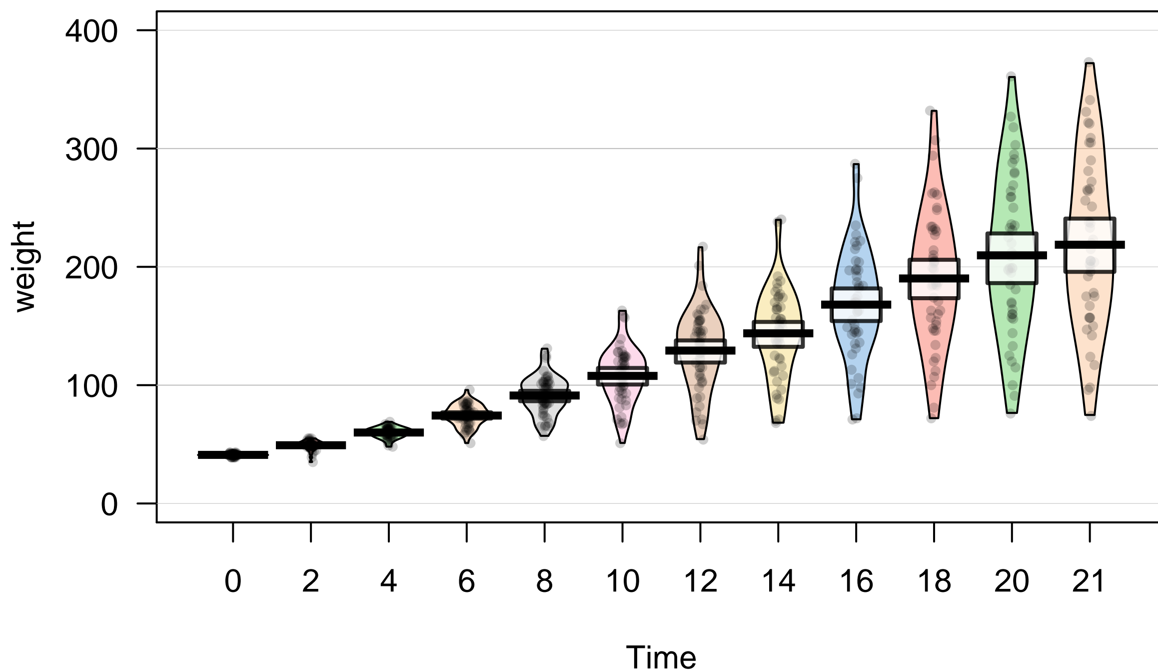
To load a package, you use the `library()` function. For example, now that we've installed the `yarrr`

package, we can load it with `library("yarrr")`:

```
# Load the yarrr package so I can use it!
#   You have to load a package in every new R session!
library("yarrr")
```

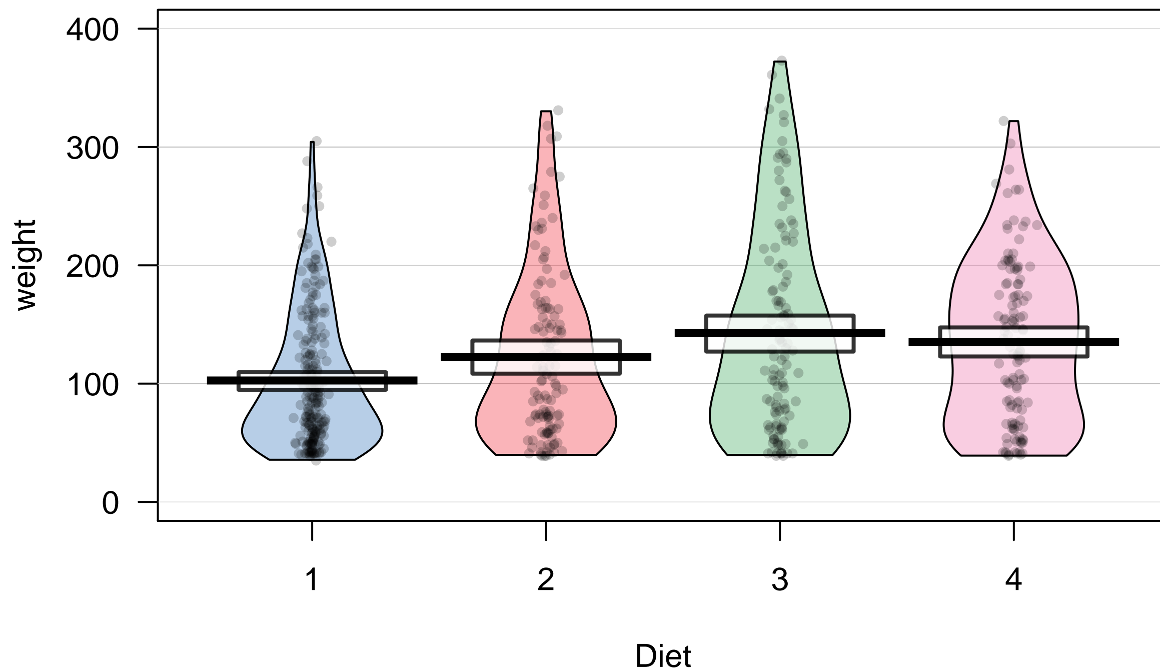
Now that you've loaded the `yarrr` package, you can use any of its functions! One of the coolest functions in this package is called `pirateplot()`. Rather than telling you what a pirateplot is, let's just make one. Run the following code chunk to make your own pirateplot. Don't worry about the specifics of the code below, you'll learn more about how all this works later. For now, just run the code and marvel at your pirateplot.

```
# Make a pirateplot using the pirateplot() function
#  from the yarrr package!

pirateplot(formula = weight ~ Time,
           data = ChickWeight,
           pal = "xmen")
```



There is one way in R to temporarily load a package without using the `library()` function. To do this, you can simply use the notation `package::function` notation. This notation simply tells R to load the package just for this one chunk of code. For example, I could use the `pirateplot` function from `yarrr` package as follows:

```
# Use the pirateplot() function without loading the yarrr package first
yarrr::pirateplot(formula = weight ~ Diet,
                  data = ChickWeight)
```



Again, you can think about the `package::function` method as a way to temporarily loading a package for a single line of code. One benefit of using the `package::function` notation is that it's immediately clear to anyone reading the code which package contains the function. However, a drawback is that if you are using a function from a package often, it forces you to constantly retype the package name. You can use whichever method makes sense for you.

2.4 Reading and writing Code

2.4.1 Code Chunks

In this book, R code is (almost) always presented in a separate gray box like this one:

```
# A code chunk

# Define a vector a as the integers from 1 to 5
a <- 1:5

# Print a
a
## [1] 1 2 3 4 5

# What is the mean of a?
mean(a)
## [1] 3
```

This is called a *code chunk*. You should always be able to copy and paste code chunks directly into R. If you copy a chunk and it does not work for you, it is most likely because the code refers to a package, function, or object that I defined in a previous chunk. If so, read back and look for a previous chunk that contains the missing definition.

2.4.2 Comments with

Lines that begin with # are comments. If you evaluate any code that starts with #, R will just ignore that line. In this book, comments will be either be literal comments that I write directly to explain code, or they will be *output* generated automatically from R. For example, in the code chunk below, you see lines starting with ##. These are the output from the previous line(s) of code. When you run the code yourself, you should see the same output in your *console*.

```
# This is a comment I wrote

1 + 2
## [1] 3

# The line above (## [1] 3) is the output from the previous code that has been 'commented out'
```

2.4.3 Element numbers in output [1]

The output you see will often start with one or more number(s) in brackets such as [1]. This is just a visual way of telling you where the numbers occur in the output. For example, in the code below, I will print a long vector containing the multiples of 2 from 0 to 100:

```
seq(from = 0, to = 100, by = 2)
##  [1]   0   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32
## [18]  34  36  38  40  42  44  46  48  50  52  54  56  58  60  62  64  66
## [35]  68  70  72  74  76  78  80  82  84  86  88  90  92  94  96  98 100
```

As you can see, the first line of the output starts with ## [1], and the next two lines start with [18] and [35]. This is just telling you that 0 is the [1]st element, 34 is the [18]th element, and 68 is the [35]th element. Sometimes this information will be helpful, but most of the time you can just ignore it.

2.5 Debugging

When you are programming, you will always, and I do mean always, make errors (also called bugs) in your code. You might misspell a function, include an extra comma, or some days…R just won't want to work with you (again, see section Why R is like a Relationship).

Debugging will always be a challenge. However, over time you'll learn which bugs are the most common and get faster and faster at finding and correcting them.

Here are the most common bugs you'll run into as you start your R journey.

2.5.1 R is not ready (>)

Another very common problem occurs when R does not seem to be responding to your code. That is, you might run some code like `mean(x)` expecting some output, but instead, nothing happens. This can be very frustrating because, rather than getting an error, just nothing happens at all. The most common reason for this is because R isn't *ready* for new code, instead, it is *waiting* for you to finish code you started earlier, but never properly finished.

Think about it this way, R can be in one of two states: it is either **Ready** (>) for new code, or it is **Waiting** (+) for you to finish old code. To see which state R is in, all you have to do is look at the symbol on the console. The `>` symbol means that R is Ready for new code – this is usually what you want to see. The `+` symbol means that R is Waiting for you to (properly) finish code you started before. If you see the `+`

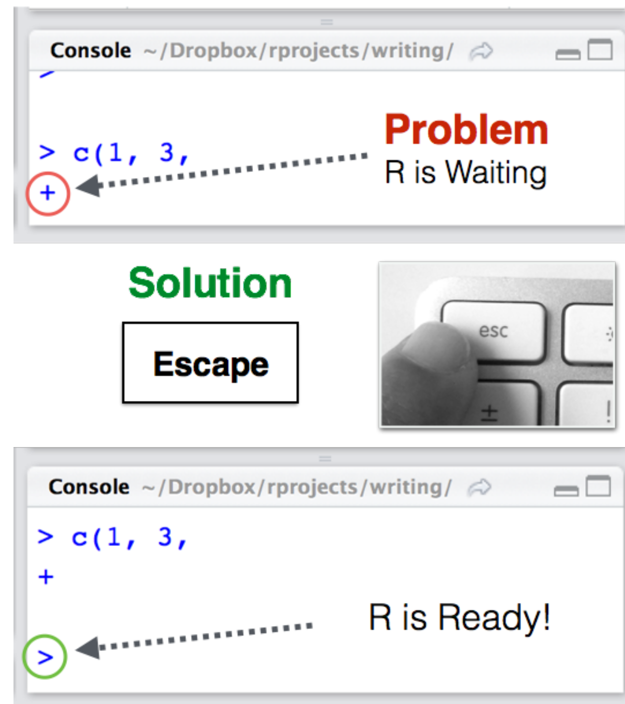


Figure 2.9: To turn R from a Waiting (+) state to a Ready (>) state, just hit Escape.

symbol, then no matter how much new code you write, R won't actually evaluate it until you finish the code you started before.

Thankfully there is an easy solution to this problem (See Figure 2.9): Just hit the escape key on your keyboard. This will cancel R's waiting state and make it Ready!

2.5.2 Misspelled object or function

If you spell an object or function incorrectly, you'll receive an error like `Error: could not find function` or `Error: object 'x' not found`.

In the code below, I'll try to take the mean of a vector `data`, but I will misspell the function `mean()`

```
data <- c(1, 4, 3, 2, 1)

# Misspelled function: should be mean(x), not meeen(x)
meeen(data)
```

Error: could not find function "meeen"

Now, I'll misspell the object `data` as `dta`:

```
# Misspelled object: should be data, not dta
mean(dta)
```

Error: object 'dta' not found

R is case-sensitive, so if you don't use the correct capitalization you'll receive an error. In the code below, I'll use `Mean()` instead of the correct version `mean()`

```
# Capitalization is wrong: should be mean(), not Mean()
Mean(data)
```

Error: could not find function "Mean"

Here is the correct version where both the object `data` and function `mean()` are correctly spelled:

```
# Correct: both the object and function are correctly spelled
mean(data)
## [1] 2.2
```

2.5.3 Punctuation problems

Another common error is having bad coding "punctuation". By that, I mean having an extra space, missing a comma, or using a comma (,) instead of a period (.). In the code below, I'll try to create a vector using periods instead of commas:

```
# Wrong: Using periods (.) instead of commas (,)
mean(c(1. 4. 2))
```

Error: unexpected numeric constant in "mean(c(1. 4."

Because I used periods instead of commas, I get the above error. Here is the correct version

```
# Correct
mean(c(1, 4, 2))
## [1] 2.3
```

If you include an extra space in the middle of the name of an object or function, you'll receive an error. In the code below, I'll accidentally write `Chick Weight` instead of `ChickWeight`:

```
# Wrong: Extra space in the ChickWeight object name
head(Chick Weight)
```

Error: unexpected symbol in "head(Chick Weight"

Because I had an extra space in the object name, I get the above error. Here is the correction:

```
# Correct:
head(ChickWeight)
```

Chapter 3

Jump In!

What's the first exercise on the first day of pirate swimming lessons? While it would be cute if they all had little inflatable pirate ships to swim around in – unfortunately this is not the case. Instead, those baby pirates take a walk off their baby planks so they can get a taste of what they're in for. Turns out, learning R is the same way. Let's jump in. In this chapter, you'll see how easy it is to calculate basic statistics and create plots in R. Don't worry if the code you're running doesn't make immediate sense – just marvel at how easy it is to do this in R!

In this section, we'll analyze a dataset called…wait for it…pirates! The dataset contains data from a survey of 1,000 pirates. The data is contained in the `yarrr` package, so make sure you've installed and loaded the package:

```
# Install the yarrr package
install.packages('yarrr')

# Load the package
library(yarrr)
```

3.1 Exploring data

Next, we'll look at the help menu for the pirates dataset using the question mark `?pirates`. When you run this, you should see a small help window open up in RStudio that gives you some information about the dataset.

```
?pirates
```

First, let's take a look at the first few rows of the dataset using the `head()` function. This will show you the first few rows of the data.

```
# Look at the first few rows of the data
head(pirates)
##      id  sex age height weight headband college tattoos tchests parrots
## 2     2 male  31    209    106      yes   JSSFP       9      11       0
## 793 793 male  25    209    104      yes    CCCC       8      27       9
## 430 430 male  26    201     99      yes    CCCC       4       7       1
## 292 292 male  29    201    102      yes    CCCC       9       2       3
## 895 895 male  27    201    103      yes    CCCC      12       1       1
## 409 409 male  28    201     97      yes    CCCC       7      10       0
##     favorite.pirate sword.type eyepatch sword.time beard.length
```

Figure 3.1: Despite what you might find at family friendly waterparks – this is NOT how real pirate swimming lessons look.

```
## 2      Jack Sparrow    cutlass        0        1.1           21
## 793        Anicetus    cutlass        1        1.1           16
## 430    Jack Sparrow    cutlass        1        0.9           14
## 292    Jack Sparrow      sabre        1        9.9           14
## 895            Hook    cutlass        1        2.3           25
## 409    Jack Sparrow    cutlass        1        1.2           15
##               fav.pixar grogg
## 2                WALL-E     9
## 793 Monsters University     8
## 430              WALL-E     9
## 292              WALL-E     6
## 895               Brave    14
## 409          Inside Out     7
```

You can look at the names of the columns in the dataset with the `names()` function

```
# What are the names of the columns?
names(pirates)
##  [1] "id"                "sex"               "age"
##  [4] "height"            "weight"            "headband"
##  [7] "college"           "tattoos"           "tchests"
## [10] "parrots"           "favorite.pirate" "sword.type"
## [13] "eyepatch"          "sword.time"        "beard.length"
## [16] "fav.pixar"         "grogg"
```

Finally, you can also view the entire dataset in a separate window using the `View()` function:

```
# View the entire dataset in a new window
View(pirates)
```

3.2 Descriptive statistics

Now let's calculate some basic statistics on the entire dataset. We'll calculate the mean age, maximum height, and number of pirates of each sex:

```
# What is the mean age?
mean(pirates$age)
## [1] 27

# What was the tallest pirate?
max(pirates$height)
## [1] 209

# How many pirates are there of each sex?
table(pirates$sex)
##
## female   male  other
##    464    490     46
```

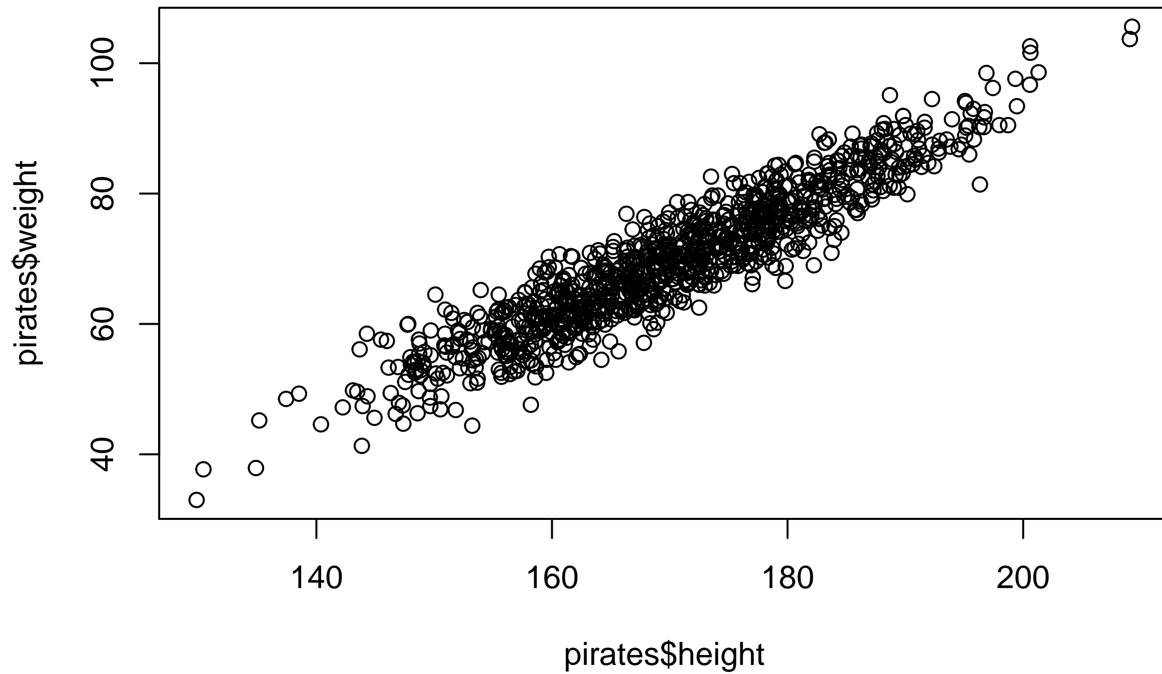
Now, let's calculate statistics for different groups of pirates. For example, the following code will use the `aggregate()` function to calculate the mean age of pirates, separately for each sex.

```
# Calculate the mean age, separately for each sex
aggregate(formula = age ~ sex,
          data = pirates,
          FUN = mean)
##      sex age
## 1 female  30
## 2   male  25
## 3  other  27
```

3.3 Plotting

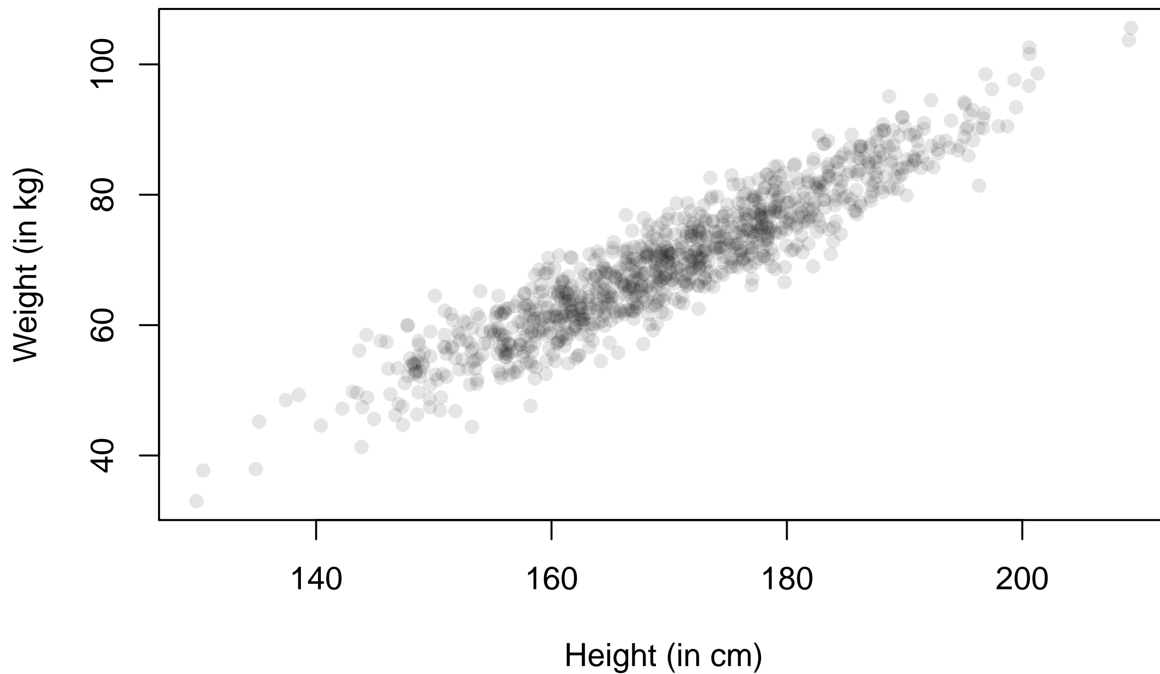
Cool stuff, now let's make a plot! We'll plot the relationship between pirate's height and weight using the `plot()` function

```
# Create scatterplot
plot(x = pirates$height,        # X coordinates
     y = pirates$weight)        # y-coordinates
```



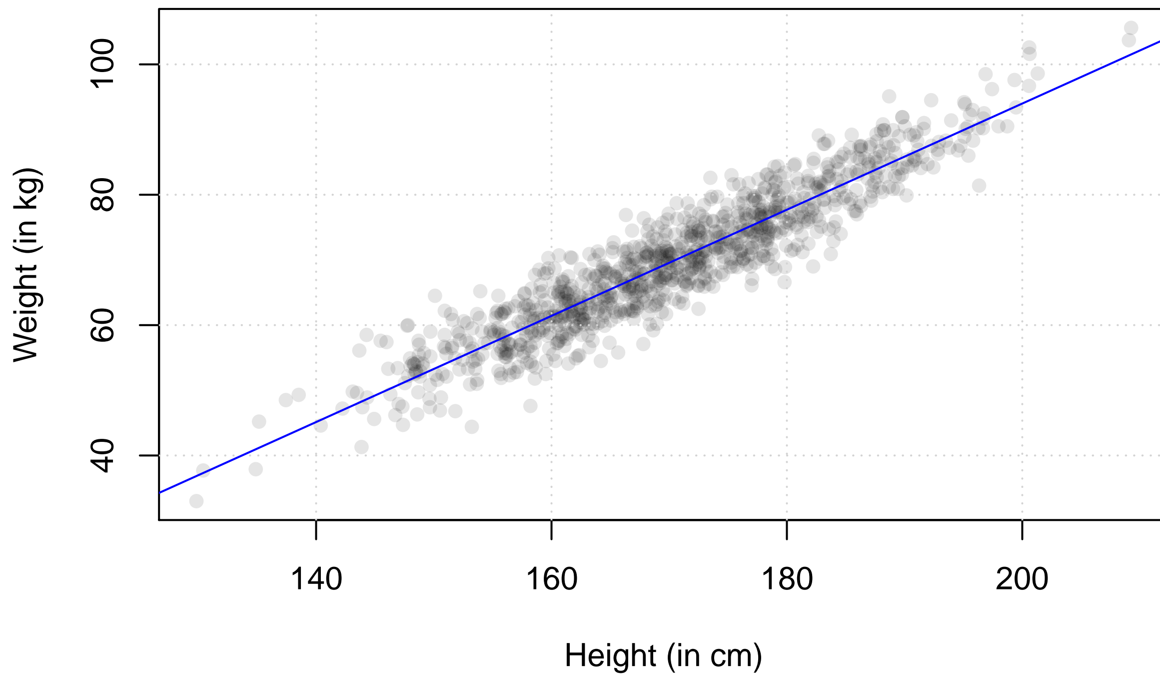
Now let's make a fancier version of the same plot by adding some customization

```
# Create scatterplot
plot(x = pirates$height,        # X coordinates
     y = pirates$weight,        # y-coordinates
     main = 'My first scatterplot of pirate data!',
     xlab = 'Height (in cm)',   # x-axis label
     ylab = 'Weight (in kg)',   # y-axis label
     pch = 16,                  # Filled circles
     col = gray(.0, .1))        # Transparent gray
```



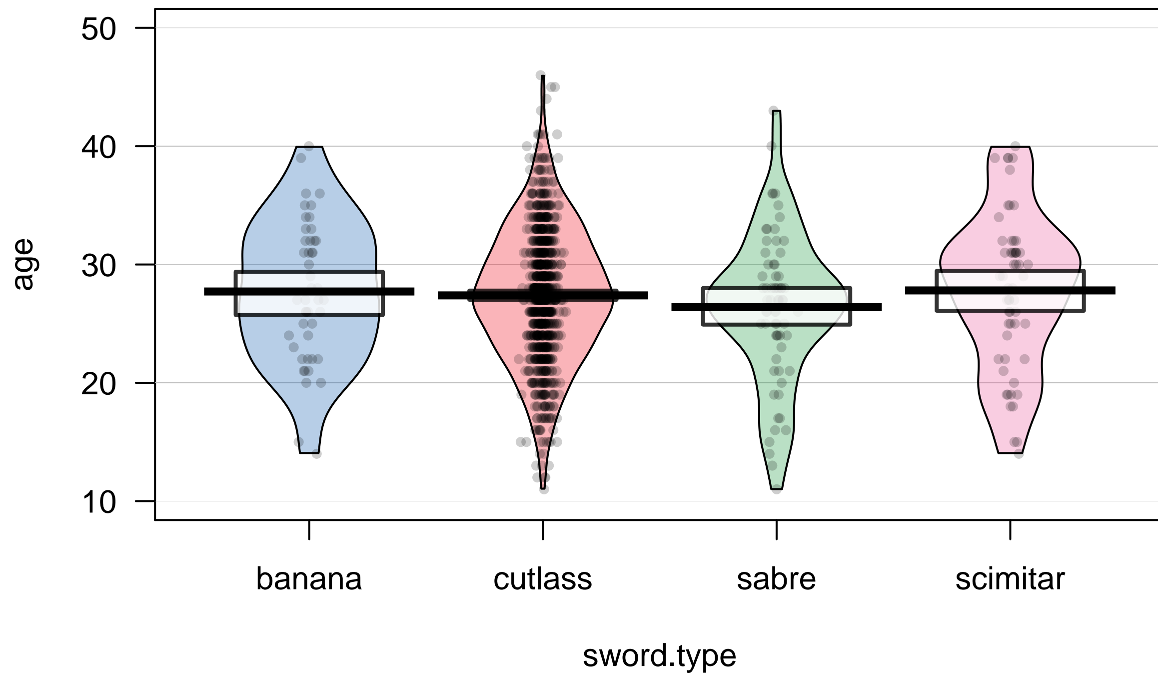
Now let's make it even better by adding gridlines and a blue regression line to measure the strength of the relationship.

```
# Create scatterplot
plot(x = pirates$height,        # X coordinates
     y = pirates$weight,        # y-coordinates
     main = 'My first scatterplot of pirate data!',
     xlab = 'Height (in cm)',   # x-axis label
     ylab = 'Weight (in kg)',   # y-axis label
     pch = 16,                  # Filled circles
     col = gray(.0, .1))        # Transparent gray

grid()        # Add gridlines

# Create a linear regression model
model <- lm(formula = weight ~ height,
            data = pirates)

abline(model, col = 'blue')      # Add regression to plot
```



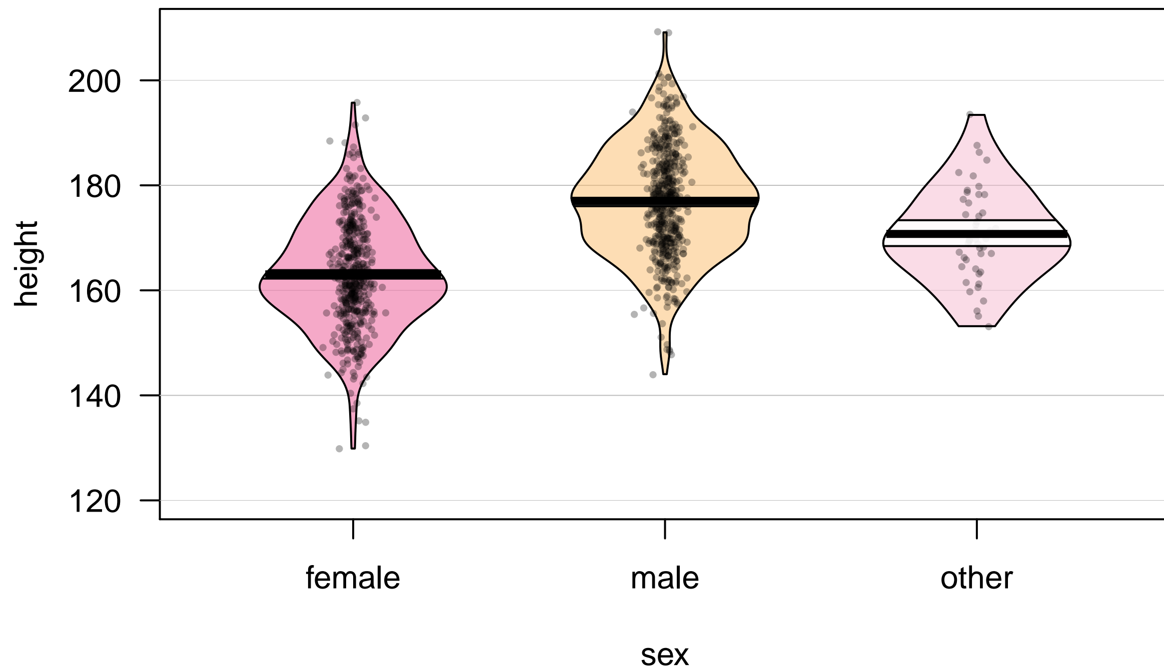
Scatterplots are great for showing the relationship between two continuous variables, but what if your independent variable is not continuous? In this case, pirateplots are a good option. Let's create a pirateplot using the `pirateplot()` function to show the distribution of pirate's age based on their favorite sword:

```
pirateplot(formula = age ~ sword.type,
           data = pirates,
           main = "Pirateplot of ages by favorite sword")
```



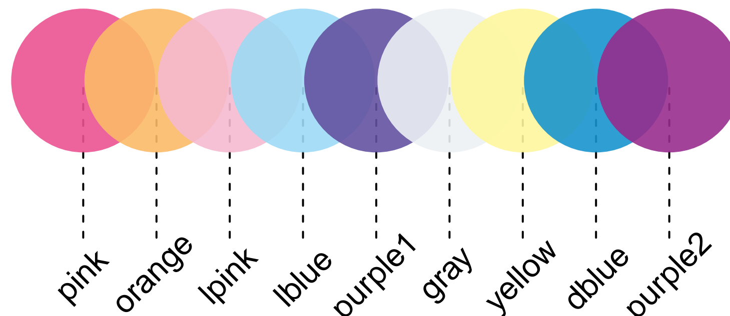
Now let's make another pirateplot showing the relationship between sex and height using a different plotting theme and the "pony" color palette:

```
pirateplot(formula = height ~ sex,                # Plot weight as a function of sex
           data = pirates,
           main = "Pirateplot of height by sex",
           pal = "pony",                          # Use the info color palette
           theme = 3)                             # Use theme 3
```



The "pony" palette is contained in the piratepal() function. Let's see where the "pony" palette comes from...

```
# Show me the pony palette!
piratepal(palette = "pony",
          plot.result = TRUE,    # Plot the result
          trans = .1)            # Slightly transparent
```



3.4 Hypothesis tests

Now, let's do some basic hypothesis tests. First, let's conduct a two-sample t-test to see if there is a significant difference between the ages of pirates who do wear a headband, and those who do not:

```
# Age by headband t-test
t.test(formula = age ~ headband,
       data = pirates,
       alternative = 'two.sided')
##
##  Welch Two Sample t-test
##
## data:  age by headband
## t = 0.4, df = 100, p-value = 0.7
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.0  1.5
## sample estimates:
##  mean in group no mean in group yes
##                28                27
```

With a p-value of 0.7259, we don't have sufficient evidence say there is a difference in the men age of pirates who wear headbands and those that do not.

Next, let's test if there a significant correlation between a pirate's height and weight using the `cor.test()` function:

```
cor.test(formula = ~ height + weight,
         data = pirates)
```

```
##
##  Pearson's product-moment correlation
##
## data:  height and weight
## t = 80, df = 1000, p-value <2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.92 0.94
## sample estimates:
##  cor
## 0.93
```

We got a p-value of `p < 2.2e-16`, that's scientific notation for `p < .00000000000000016` – which is pretty much 0. Thus, we'd conclude that there is a significant (positive) relationship between a pirate's height and weight.

Now, let's do an ANOVA testing if there is a difference between the number of tattoos pirates have based on their favorite sword

```
# Create tattoos model
tat.sword.lm <- lm(formula = tattoos ~ sword.type,
                   data = pirates)

# Get ANOVA table
anova(tat.sword.lm)
## Analysis of Variance Table
##
## Response: tattoos
##             Df Sum Sq Mean Sq F value Pr(>F)
## sword.type   3   1588     529    54.1 <2e-16 ***
## Residuals  996   9743      10
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Sure enough, we see another very small p-value of `p < 2.2e-16`, suggesting that the number of tattoos pirate's have are different based on their favorite sword.

3.5 Regression analysis

Finally, let's run a regression analysis to see if a pirate's age, weight, and number of tattoos (s)he has predicts how many treasure chests he/she's found:

```
# Create a linear regression model: DV = tchests, IV = age, weight, tattoos
tchests.model <- lm(formula = tchests ~ age + weight + tattoos,
                    data = pirates)

# Show summary statistics
summary(tchests.model)
##
## Call:
## lm(formula = tchests ~ age + weight + tattoos, data = pirates)
##
## Residuals:
##    Min     1Q Median     3Q    Max
```

```
## -33.30 -15.83  -6.86   8.41 119.97
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   5.1908     7.1844    0.72     0.47
## age           0.7818     0.1344    5.82    8e-09 ***
## weight       -0.0901     0.0718   -1.25     0.21
## tattoos       0.2540     0.2255    1.13     0.26
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 24 on 996 degrees of freedom
## Multiple R-squared:  0.0406, Adjusted R-squared:  0.0377
## F-statistic:   14 on 3 and 996 DF,  p-value: 5.75e-09
```

It looks like the only significant predictor of the number of treasure chests that a pirate has found is his/her age. There does not seem to be significant effect of weight or tattoos.

3.6 Bayesian Statistics

Now, let's repeat some of our previous analyses with Bayesian versions. First we'll install and load the `BayesFactor` package which contains the Bayesian statistics functions we'll use:

```
# Install and load the BayesFactor package
install.packages('BayesFactor')
library(BayesFactor)
```

Now that the packages is installed and loaded, we're good to go! Let's do a Bayesian version of our earlier t-test asking if pirates who wear a headband are older or younger than those who do not.

```
# Bayesian t-test comparing the age of pirates with and without headbands
ttestBF(formula = age ~ headband,
        data = pirates)
## Bayes factor analysis
## --------------
## [1] Alt., r=0.707 : 0.12 ±0%
##
## Against denominator:
##   Null, mu1-mu2 = 0
## ---
## Bayes factor type: BFindepSample, JZS
```

It looks like we got a Bayes factor of 0.12 which is strong evidence *for* the null hypothesis (that the mean age does not differ between pirates with and without headbands)

3.7 Wasn't that easy?!

Wait…wait…WAIT! Did you seriously just calculate descriptive statistics, a t-test, an ANOVA, and a regression, create a scatterplot and a pirateplot, AND do both a Bayesian t-test and regression analysis. Yup. Imagine how long it would have taken to explain how to do all that in SPSS. And while you haven't really learned how R works yet, I'd bet my beard that you could easily alter the previous code to do lots of

other analyses. Of course, don't worry if some or all of the previous code didn't make sense. Soon…it will all be clear.

Now that you've jumped in, let's learn how to swim.

Chapter 4

The Basics

If you're like most people, you think of R as a statistics program. However, while R is definitely the coolest, most badass, pirate-y way to conduct statistics – it's not really a program. Rather, it's a programming *language* that was written by and for statisticians. To learn more about the history of R…just…you know…Google it.

In this chapter, we'll go over the basics of the R language and the RStudio programming environment.

4.1 The command-line (Console)

R code, on its own, is just text. You can write R code in a new script within R or RStudio, or in any text editor. Hell, you can write R code on Twitter if you want. However, just writing the code won't do the whole job – in order for your code to be executed (aka, interpreted) you need to send it to R's *command-line interpreter*. In RStudio, the command-line interpreter is called the Console.

In R, the command-line interpreter starts with the `>` symbol. This is called the **prompt**. Why is it called the prompt? Well, it's "prompting" you to feed it with some R code. The fastest way to have R evaluate code is to type your R code directly into the command-line interpreter. For example, if you type `1+1` into the interpreter and hit enter you'll see the following

```
1+1
## [1] 2
```

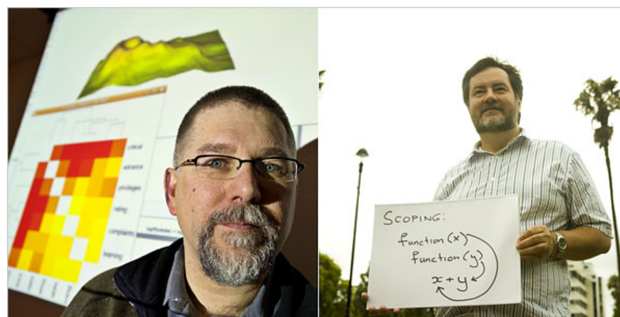


Figure 4.1: Ross Ihaka and Robert Gentlemen. You have these two pirates to thank for creating R! You might not think much of them now, but by the end of this book there's a good chance you'll be dressing up as one of them on Halloween.

Figure 4.2: Yep. R is really just a fancy calculator. This R programming device was found on a shipwreck on the Bodensee in Germany. I stole it from a museum and made a pretty sweet plot with it. But I don't want to show it to you.

This is the **console**

Console ~/Dropbox/manuscripts/FFTrees_man/

```
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
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 brow
Type 'q()' to quit R.

[Workspace loaded from ~/Dropbo

> 1+1
[1] 2
> |
```

Here is the **command-line**. You can type here to get an immediate response

Figure 4.3: You can always type code directly into the command line to get an immediate response.

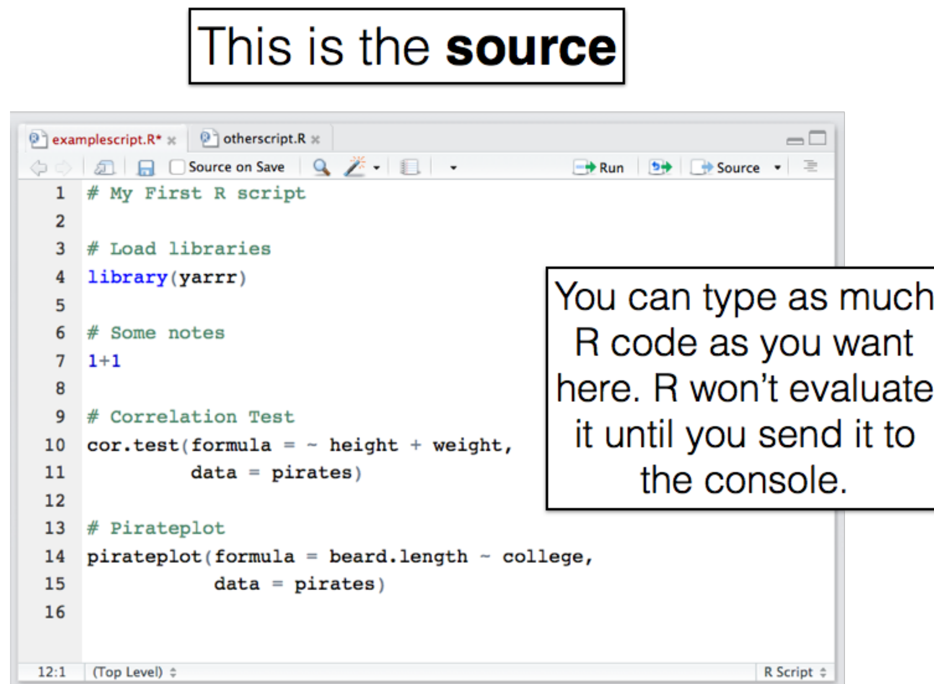


Figure 4.4: Here's how a new script looks in the editor window on RStudio. The code you type won't be executed until you send it to the console.

As you can see, R returned the (thankfully correct) value of 2. You'll notice that the console also returns the text [1]. This is just telling you you the index of the value next to it. Don't worry about this for now, it will make more sense later. As you can see, R can, thankfully, do basic calculations. In fact, at its heart, R is technically just a fancy calculator. But that's like saying Michael Jordan is *just* a fancy ball bouncer or Donald Trump is *just* an orange with a dead fox on his head. It (and they), are much more than that.

4.2 Writing R scripts in an editor

There are certainly many cases where it makes sense to type code directly into the console. For example, to open a help menu for a new function with the ? command, to take a quick look at a dataset with the `head()` function, or to do simple calculations like `1+1`, you should type directly into the console. However, the problem with writing all your code in the console is that nothing that you write will be saved. So if you make an error, or want to make a change to some earlier code, you have to type it all over again. Not very efficient. For this (and many more reasons), you'll should write any important code that you want to save as an R script. An R script is just a bunch of R code in a single file. You can write an R script in any text editor, but you should save it with the `.R` suffix to make it clear that it contains R code.} in an editor.

In RStudio, you'll write your R code in the...wait for it...*Source* window. To start writing a new R script in RStudio, click File – New File – R Script.

Shortcut! To create a new script in R, you can also use the command–shift–N shortcut on Mac. I don't know what it is on PC...and I don't want to know.

When you open a new script, you'll see a blank page waiting for you to write as much R code as you'd like. In Figure 4.4, I have a new script called `examplescript` with a few random calculations.

You can have several R scripts open in the source window in separate tabs (like I have above).

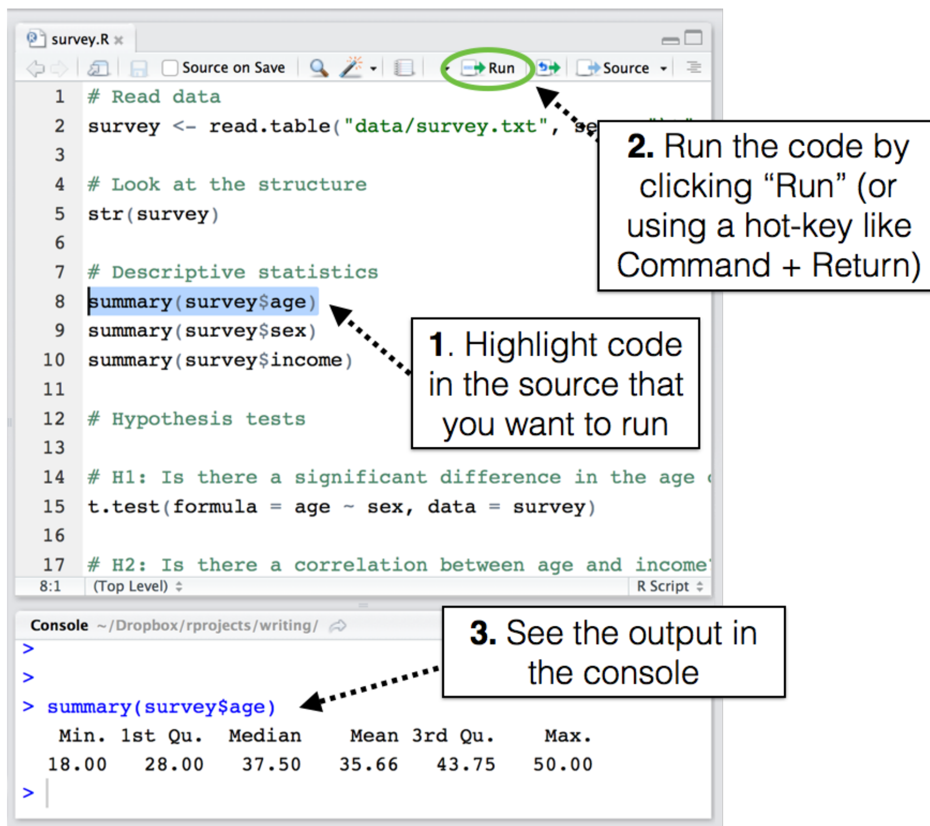


Figure 4.5: To evaluate code from the source, highlight it and run it.

4.2.1 Send code from an source to the console

When you type code into an R script, you'll notice that, unlike typing code into the Console, nothing happens. In order for R to interpret the code, you need to send it from the Editor to the Console. There are a few ways to do this, here are the three most common ways:

1. Copy the code from the Editor (or anywhere that has valid R code), and paste it into the Console (using Command–V).
2. Highlight the code you want to run (with your mouse or by holding Shift), then use the Command–Return shortcut (see Figure 4.6).
3. Place the cursor on a single line you want to run, then use the Command–Return shortcut to run just that line.

99% of the time, I use method 2, where I highlight the code I want, then use the Command–Return shortcut . However, method 3 is great for trouble-shooting code line-by-line.

4.3 A brief style guide: Commenting and spacing

Like all programming languages, R isn't just meant to be read by a computer, it's also meant to be read by other humans – or very well-trained dolphins. For this reason, it's important that your code looks nice and is understandable to other people and your future self. To keep things brief, I won't provide a complete style guide – instead I'll focus on the two most critical aspects of good style: commenting and spacing.

Figure 4.6: Ah...the Command–Return shortcut (Control–Enter on PC) to send highlighted code from the Editor to the Console. Get used to this shortcut people. You're going to be using this a lot

Figure 4.7: As Stan discovered in season six of South Park, your future self is a lazy, possibly intoxicated moron. So do your future self a favor and make your code look nice. Also maybe go for a run once in a while.

4.3.1 Commenting code with the # (pound) sign

Comments are completely ignored by R and are just there for whomever is reading the code. You can use comments to explain what a certain line of code is doing, or just to visually separate meaningful chunks of code from each other. Comments in R are designated by a # (pound) sign. Whenever R encounters a # sign, it will ignore **all** the code after the # sign on that line. Additionally, in most coding editors (like RStudio) the editor will display comments in a separate color than standard R code to remind you that it's a comment:

Here is an example of a short script that is nicely commented. Try to make your scripts look like this!

```
# Author: Pirate Jack
# Title: My nicely commented R Script
# Date: None today :(

# Step 1: Load the yarrr package
library(yarrr)

# Step 2: See the column names in the movies dataset
names(movies)

# Step 3: Calculations

# What percent of movies are sequels?
mean(movies$sequel, na.rm = T)

# How much did Pirate's of the Caribbean: On Stranger Tides make?
movies$revenue.all[movies$name == 'Pirates of the Caribbean: On Stranger Tides']
```

I cannot stress enough how important it is to comment your code! Trust me, even if you don't plan on sharing your code with anyone else, keep in mind that your future self will be reading it in the future.

4.3.2 Spacing

Howwouldyouliketoreadabookiftherewerenospacesbetweenwords? I'mguessingyouwouldn't. Soeverytimeyouwritecodewithoutproperspacing,rememberthissentence.

Commenting isn't the only way to make your code legible. It's important to make appropriate use of spaces and line breaks. For example, I include spaces between arithmetic operators (like =, + and -) and after commas (which we'll get to later). For example, look at the following code:

```
# Shitty looking code
a<-(100+3)-2
mean(c(a/100,642564624.34))
t.test(formula=revenue.all~sequel,data=movies)
plot(x=movies$budget,y=movies$dvd.usa,main="myplot")
```

That code looks like shit. Don't write code like that. It makes my eyes hurt. Now, let's use some liberal amounts of commenting and spacing to make it look less shitty.

```
# Some meaningless calculations. Not important

a <- (100 + 3) - 2
mean(c(a / 100, 642564624.34))

# t.test comparing revenue of sequels v non-sequels
```

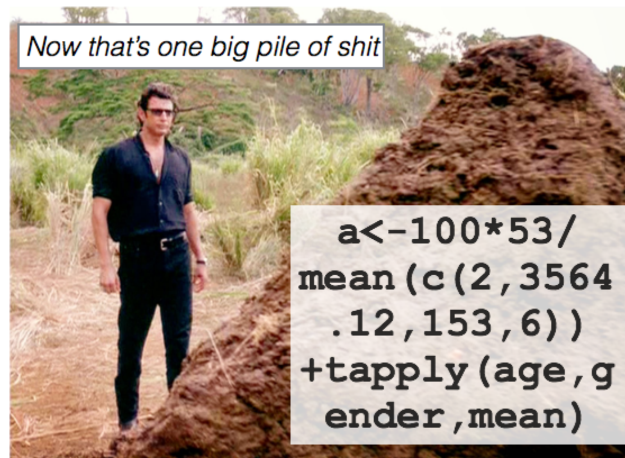


Figure 4.8: Don't make your code look like what a sick Triceratops with diarrhea left behind for Jeff Goldblum.

```
t.test(formula = revenue.all ~ sequel,
       data = movies)

# A scatterplot of budget and dvd revenue.
#  Hard to see a relationship

plot(x = movies$budget,
     y = movies$dvd.usa,
     main = "myplot")
```

See how much better that second chunk of code looks? Not only do the comments tell us the purpose behind the code, but there are spaces and line-breaks separating distinct elements.

There are a lot more aspects of good code formatting. For a list of recommendations on how to make your code easier to follow, check out Google's own company R Style guide at https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml

4.4 Objects and functions

To understand how R works, you need to know that R revolves around two things: objects and functions. Almost everything in R is either an object or a function. In the following code chunk, I'll define a simple object called `tattoos` using a function `c()`:

```
# 1: Create a vector object called tattoos
tattoos <- c(4, 67, 23, 4, 10, 35)

# 2: Apply the mean() function to the tattoos object
mean(tattoos)
## [1] 24
```

What is an object? An object is a thing – like a number, a dataset, a summary statistic like a mean or standard deviation, or a statistical test. Objects come in many different shapes and sizes in R. There are simple objects like *scalars* which represent single numbers, **vectors** (like our `tattoos` object above) which represent several numbers, more complex objects like **dataframes** which represent tables of data, and even

more complex objects like **hypothesis tests** or **regression** which contain all sorts of statistical information.

Different types of objects have different *attributes*. For example, a vector of data has a length attribute (i.e.; how many numbers are in the vector), while a hypothesis test has many attributes such as a test-statistic and a p-value. Don't worry if this is a bit confusing now – it will all become clearer when you meet these new objects in person in later chapters. For now, just know that objects in R are things, and different objects have different attributes.

What is a function? A function is a *procedure* that typically takes one or more objects as arguments (aka, inputs), does something with those objects, then returns a new object. For example, the `mean()` function we used above takes a vector object, like `tattoos`, of numeric data as an argument, calculates the arithmetic mean of those data, then returns a single number (a scalar) as a result.A great thing about R is that you can easily create your own functions that do whatever you want – but we'll get to that much later in the book. Thankfully, R has hundreds (thousands?) of built-in functions that perform most of the basic analysis tasks you can think of.

99% of the time you are using R, you will do the following: 1) Define objects. 2) Apply functions to those objects. 3) Repeat!. Seriously, that's about it. However, as you'll soon learn, the hard part is knowing how to define objects they way you want them, and knowing which function(s) will accomplish the task you want for your objects.

4.4.1 Numbers versus characters

For the most part, objects in R come in one of two flavors: **numeric** and **character**. It is very important to keep these two separate as certain functions, like `mean()`, and `max()` will only work for numeric objects, while functions like `grep()` and `strtrim()` only work for character objects.

A numeric object is just a number like `1`, `10` or `3.14`. You don't have to do anything special to create a numeric object, just type it like you were using a calculator.

```
# These are all numeric objects
1
10
3.14
```

A **character** object is a name like `"Madisen"`, `"Brian"`, or `"University of Konstanz"`. To specify a character object, you need to include quotation marks `""` around the text.

```
# These are all character objects
"Madisen"
"Brian"
"10"
```

If you try to perform a function or operation meant for a numeric object on a character object (and vice-versa), R will yell at you. For example, here's what happens when I try to take the mean of the two character objects `"1"` and `"10"`:

```
# This will return an error because the arguments are not numeric!
mean(c("1", "10"))
```

Warning message: argument is not numeric or logical, returning NA

If I make sure that the arguments are numeric (by not including the quotation marks), I won't receive the error:

```
# This is ok!
mean(c(1, 10))
## [1] 5.5
```

4.4.2 Creating new objects with <-

By now you know that you can use R to do simple calculations. But to really take advantage of R, you need to know how to create and manipulate objects. All of the data, analyses, and even plots, you use and create are, or can be, saved as objects in R. For example the `movies` dataset which we've used before is an object stored in the `yarrr` package. This object was defined in the `yarrr` package with the name `movies`. When you loaded the `yarrr` package with the `library('yarrr')` command, you told R to give you access to the `movies` object. Once the object was loaded, we could use it to calculate descriptive statistics, hypothesis tests, and to create plots.

To create new objects in R, you need to do *object assignment*. Object assignment is our way of storing information, such as a number or a statistical test, into something we can easily refer to later. This is a pretty big deal. Object assignment allows us to store data objects under relevant names which we can then use to slice and dice specific data objects anytime we'd like to.

To do an assignment, we use the almighty `<-` operator called *assign* To assign something to a new object (or to change an existing object), use the notation `object <- ...`}, where `object` is the new (or updated) object, and `...` is whatever you want to store in `object`. Let's start by creating a very simple object called `a` and assigning the value of 100 to it:

Good object names strike a balance between being easy to type (i.e.; short names) and interpret. If you have several datasets, it's probably not a good idea to name them `a`, `b`, `c` because you'll forget which is which. However, using long names like `March2015Group1OnlyFemales` will give you carpal tunnel syndrome.

```
# Create a new object called a with a value of 100
a <- 100
```

Once you run this code, you'll notice that R doesn't tell you anything. However, as long as you didn't type something wrong, R should now have a new object called `a` which contains the number 100. If you want to see the value, you need to call the object by just executing its name. This will print the value of the object to the console:

```
# Print the object a
a
## [1] 100
```

Now, R will print the value of `a` (in this case 100) to the console. If you try to evaluate an object that is not yet defined, R will return an error. For example, let's try to print the object `b` which we haven't yet defined:

```
b
```

Error: object 'b' not found

As you can see, R yelled at us because the object `b` hasn't been defined yet.

Once you've defined an object, you can combine it with other objects using basic arithmetic. Let's create objects `a` and `b` and play around with them.

```
a <- 1
b <- 100

# What is a + b?
a + b
## [1] 101

# Assign a + b to a new object (c)
c <- a + b

# What is c?
```

```
c
## [1] 101
```

4.4.2.1 To change an object, you must assign it again!

Normally I try to avoid excessive emphasis, but because this next sentence is so important, I have to just go for it. Here it goes…

To change an object, you *must* assign it again!

No matter what you do with an object, if you don't assign it again, it won't change. For example, let's say you have an object `z` with a value of 0. You'd like to add 1 to `z` in order to make it 1. To do this, you might want to just enter `z + 1` – but that won't do the job. Here's what happens if you **don't** assign it again:

```
z <- 0
z + 1
## [1] 1
```

Ok! Now let's see the value of `z`

```
z
## [1] 0
```

Damn! As you can see, the value of z is still 0! What went wrong? Oh yeah…

To change an object, you *must* assign it again!

The problem is that when we wrote `z + 1` on the second line, R thought we just wanted it to calculate and print the value of `z + 1`, without storing the result as a new `z` object. If we want to actually update the value of `z`, we need to reassign the result back to `z` as follows:

```
z <- 0
z <- z + 1  # Now I'm REALLY changing z
z
## [1] 1
```

Phew, z is now 1. Because we used assignment, z has been updated. About freaking time.

4.4.3 How to name objects

You can create object names using any combination of letters and a few special characters (like `.` and `_`). Here are some valid object names

```
# Valid object names
group.mean <- 10.21
my.age <- 32
FavoritePirate <- "Jack Sparrow"
sum.1.to.5 <- 1 + 2 + 3 + 4 + 5
```

All the object names above are perfectly valid. Now, let's look at some examples of *invalid* object names. These object names are all invalid because they either contain spaces, start with numbers, or have invalid characters:

```
# Invalid object names!
famale ages <- 50 # spaces
5experiment <- 50 # starts with a number
a! <- 50 # has an invalid character
```

Figure 4.9: Like a text message, you should probably watch your use of capitalization in R.

If you try running the code above in R, you will receive a warning message starting with

Error: unexpected symbol

. Anytime you see this warning in R, it almost always means that you have a naming error of some kind.

4.4.3.1 R is case-sensitive!

Like English, R is case-sensitive – it R treats capital letters differently from lower-case letters. For example, the four following objects `Plunder`, `plunder` and `PLUNDER` are totally different objects in R:

```
# These are all different objects
Plunder <- 1
plunder <- 100
PLUNDER <- 5
```

I try to avoid using too many capital letters in object names because they require me to hold the shift key. This may sound silly, but you'd be surprised how much easier it is to type `mydata` than `MyData` 100 times.

4.4.4 Example: Pirates of The Caribbean

Let's do a more practical example – we'll define an object called `blackpearl.usd` which has the global revenue of Pirates of the Caribbean: Curse of the Black Pearl in U.S. dollars. A quick Google search showed me that the revenue was $634,954,103. I'll create the new object using assignment:

```
blackpearl.usd <- 634954103
```

Now, my fellow European pirates might want to know how much this is in Euros. Let's create a new object called `{blackpearl.eur` which converts our original value to Euros by multiplying the original amount by 0.88 (assuming 1 USD = 0.88 EUR)

```
blackpearl.eur <- blackpearl.usd * 0.88
blackpearl.eur
## [1] 5.6e+08
```

It looks like the movie made 558,759,611 in Euros. Not bad. Now, let's see how much more Pirates of the Caribbean 2: Dead Man's Chest made compared to "Curse of the Black Pearl." Another Google search uncovered that Dead Man's Chest made $1,066,215,812 (that wasn't a mistype, the freaking movie made over a billion dollars).

```
deadman.usd <- 1066215812
```

Now, I'll divide `deadman.usd` by `blackpearl.usd`:

```
deadman.usd / blackpearl.usd
## [1] 1.7
```

It looks like "Dead Man's Chest" made 168% as much as "Curse of the Black Pearl" - not bad for two movies based off of a ride from Disneyland.

4.5 Test your R might!

1. Create a new R script. Using comments, write your name, the date, and "Testing my Chapter 2 R Might" at the top of the script. Write your answers to the rest of these exercises on this script, and be sure to copy and paste the original questions using comments! Your script should **only** contain valid R code and comments.

2. Which (if any) of the following objects names is/are invalid?

```
thisone <- 1
THISONE <- 2
1This <- 3
this.one <- 4
This.1 <- 5
ThIS.....ON...E <- 6
This!On!e <- 7
lkjasdfkjsdf <- 8
```

3. 2015 was a good year for pirate booty - your ship collected 100,800 gold coins. Create an object called `gold.in.2015` and assign the correct value to it.

4. Oops, during the last inspection we discovered that one of your pirates Skippy McGee hid 800 gold coins in his underwear. Go ahead and add those gold coins to the object `gold.in.2015`. Next, create an object called `plank.list` with the name of the pirate thief.

5. Look at the code below. What will R return after the third line? Make a prediction, then test the code yourself.

```
a <- 10
a + 10
a
```

Chapter 5

Scalars and vectors

```
# Crew information
captain.name <- "Jack"
captain.age <- 33

crew.names <- c("Heath", "Vincent", "Maya", "Becki")
crew.ages <- c(19, 35, 22, 44)
crew.sex <- c(rep("M", times = 2), rep("F", times = 2))
crew.ages.decade <- crew.ages / 10

# Earnings over first 10 days at sea
days <- 1:10
gold <- seq(from = 10, to = 100, by = 10)
silver <- rep(50, times = 10)
total <- gold + silver
```

People are not objects. But R is full of them. Here are some of the basic ones.

5.1 Scalars

The simplest object type in R is a **scalar**. A scalar object is just a single value like a number or a name. In the previous chapter we defined several scalar objects. Here are examples of numeric scalars:

```
# Examples of numeric scalers
a <- 100
b <- 3 / 100
c <- (a + b) / b
```

Scalars don't have to be numeric, they can also be **characters** (also known as strings). In R, you denote characters using quotation marks. Here are examples of character scalars:

```
# Examples of character scalers
d <- "ship"
e <- "cannon"
f <- "Do any modern armies still use cannons?"
```

As you can imagine, R treats numeric and character scalars differently. For example, while you can do basic arithmetic operations on numeric scalars – they won't work on character scalars. If you try to perform numeric operations (like addition) on character scalars, you'll get an error like this one:

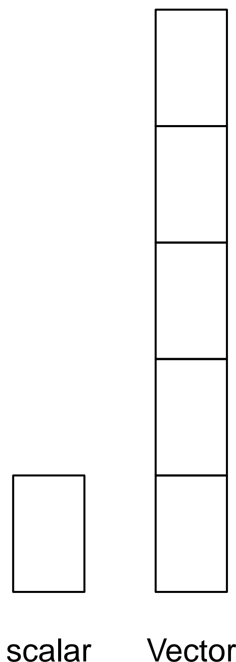


Figure 5.1: Visual depiction of a scalar and vector. Deep shit. Wait until we get to matrices - you're going to lose it.

```
a <- "1"
b <- "2"
a + b
```

Error in a + b: non-numeric argument to binary operator

If you see an error like this one, it means that you're trying to apply numeric operations to character objects. That's just sick and wrong.

5.2 Vectors

Now let's move onto `vectors`. A vector object is just a combination of several scalars stored as a single object. For example, the numbers from one to ten could be a vector of length 10, and the characters in the English alphabet could be a vector of length 26. Like scalars, vectors can be either numeric or character (but not both!).

There are many ways to create vectors in R. Here are the methods we will cover in this chapter:

Table 5.1: Functions to create vectors.

Function	Example	Result
`c(a, b, ...)`	`c(1, 5, 9)`	1, 5, 9
`a:b`	`1:5`	1, 2, 3, 4, 5
`seq(from, to, by, length.out)`	`seq(from = 0, to = 6, by = 2)`	0, 2, 4, 6
`rep(x, times, each, length.out)`	`rep(c(7, 8), times = 2, each = 2)`	7, 7, 8, 8, 7, 7, 8, 8

The simplest way to create a vector is with the `c()` function. The c here stands for concatenate, which means "bring them together". The `c()` function takes several scalars as arguments, and returns a vector containing those objects. When using c(), place a comma in between the objects (scalars or vectors) you want to combine:

Let's use the `c()` function to create a vector called `a` containing the integers from 1 to 5.

```
# Create an object a with the integers from 1 to 5
```

Figure 5.2: This is not a pipe. It is a character vector.

```
char.vec <- c("Ceci", "nest", "pas", "une", "pipe")
char.vec
## [1] "Ceci" "nest" "pas"  "une"  "pipe"
```

While the `c()` function is the most straightforward way to create a vector, it's also one of the most tedious. For example, let's say you wanted to create a vector of all integers from 1 to 100. You definitely don't want to have to type all the numbers into a c() operator. Thankfully, R has many simple built-in functions for generating numeric vectors. Let's start with three of them: `a:b`, `seq()`, and `rep()`:

5.2.1 a:b

The `a:b` function takes two numeric scalars `a` and `b` as arguments, and returns a vector of numbers from the starting point `a` to the ending point `b` in steps of 1.

Here are some examples of the `a:b` function in action. As you'll see, you can go backwards or forwards, or make sequences between non-integers:

```
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
10:1
##  [1] 10  9  8  7  6  5  4  3  2  1
2.5:8.5
## [1] 2.5 3.5 4.5 5.5 6.5 7.5 8.5
```

5.2.2 seq()

Argument	Definition
`from`	The start of the sequence
`to`	The end of the sequence
`by`	The step-size of the sequence
`length.out`	The desired length of the final sequence (only use if you don't specify `by`)

The `seq()` function is a more flexible version of `a:b`. Like `a:b`, `seq()` allows you to create a sequence from a starting number to an ending number. However, `seq()`, has additional arguments that allow you to specify either the size of the steps between numbers, or the total length of the sequence:

The `seq()` function has two new arguments `by` and `length.out`. If you use the `by` argument, the sequence will be in steps of the input to the `by` argument:

```
# Create the numbers from 1 to 10 in steps of 1
```

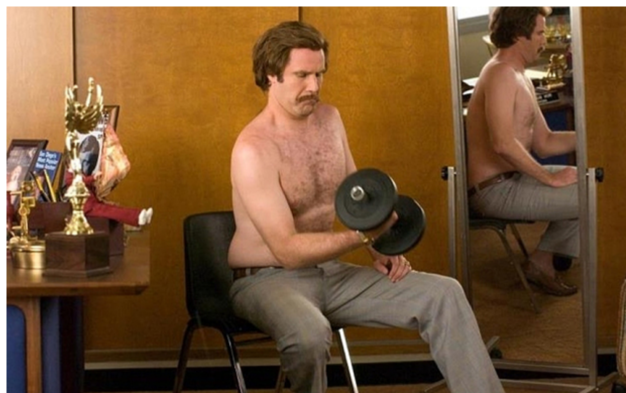


Figure 5.3: Not a good depiction of a rep in R.

```
# 3 numbers from 0 to 100
seq(from = 0, to = 100, length.out = 3)
## [1]   0  50 100
```

5.2.3 rep()

Argument	Definition
`x`	A scalar or vector of values to repeat
`times`	The number of times to repeat x
`each`	The number of times to repeat each value within x
`length.out`	The desired length of the final sequence

The `rep()` function allows you to repeat a scalar (or vector) a specified number of times, or to a desired length. Let's do some reps.

```
rep(x = 3, times = 10)
##  [1] 3 3 3 3 3 3 3 3 3 3
rep(x = c(1, 2), each = 3)
## [1] 1 1 1 2 2 2
rep(x = 1:3, length.out = 10)
##  [1] 1 2 3 1 2 3 1 2 3 1
```

As you can see, you can can include an `a:b` call within a `rep()`!

You can even combine the `times` and `each` arguments within a single `rep()` function. For example, here's how to create the sequence {1, 1, 2, 2, 3, 3, 1, 1, 2, 2, 3, 3} with one call to `rep()`:

```
rep(x = 1:3, each = 2, times = 2)
##  [1] 1 1 2 2 3 3 1 1 2 2 3 3
```

Warning! Vectors contain either numbers or characters, not both

A vector can only contain one type of scalar: either numeric or character. If you try to create a vector with numeric and character scalars, then R will convert *all* of the numeric scalars to characters. In the next code chunk, I'll create a new vector called `my.vec` that contains a mixture of numeric and character scalars.

```
my.vec <- c("a", 1, "b", 2, "c", 3)
my.vec
## [1] "a" "1" "b" "2" "c" "3"
```

As you can see from the output, `my.vec` is stored as a character vector where all the numbers are converted to characters.

5.3 Generating random data

Because R is a language built for statistics, it contains many functions that allow you generate random data – either from a vector of data that you specify (like Heads or Tails from a coin), or from an established *probability distribution*, like the Normal or Uniform distribution.

In the next section we'll go over the standard `sample()` function for drawing random values from a vector. We'll then cover some of the most commonly used probability distributions: Normal and Uniform.

5.3.1 sample()

Argument	Definition
`x`	A vector of outcomes you want to sample from. For example, to simulate coin flips, you'd enter `x = c("H", "T")`
`size`	The number of samples you want to draw. The default is the length of `x`.
`replace`	Should sampling be done with replacement? If FALSE (the default value), then each outcome in `x` can only be drawn once. If TRUE, then each outcome in `x` can be drawn multiple times.
`prob`	A vector of probabilities of the same length as `x` indicating how likely each outcome in `x` is. The vector of probabilities you give as an argument should add up to one. If you don't specify the `prob` argument, all outcomes will be equally likely.

The `sample()` function allows you to draw random samples of elements (scalars) from a vector. For example, if you want to simulate the 100 flips of a fair coin, you can tell the sample function to sample 100 values from the vector ["Heads", "Tails"]. Or, if you need to randomly assign people to either a "Control" or "Test" condition in an experiment, you can randomly sample values from the vector ["Control", "Test"]:

Let's use `sample()` to draw 10 samples from a vector of integers from 1 to 10.

```
# From the integers 1:10, draw 5 numbers
sample(x = 1:10, size  = 5)
## [1] 2 1 5 3 9
```

5.3.1.1 replace = TRUE

If you don't specify the `replace` argument, R will assume that you are sampling *without* replacement. In other words, each element can only be sampled once. If you want to sample with replacement, use the `replace = TRUE` argument:

Think about replacement like drawing balls from a bag. Sampling *with* replacement (`replace = TRUE`) means that each time you draw a ball, you return the ball back into the bag before drawing another ball.

Sampling *without* replacement (`replace = FALSE`) means that after you draw a ball, you remove that ball from the bag so you can never draw it again.

```
# Draw 30 samples from the integers 1:5 with replacement
sample(x = 1:5, size = 10, replace = TRUE)
##  [1] 2 3 5 2 3 5 4 4 2 1
```

If you try to draw a large sample from a vector *without* replacement, R will return an error because it runs out of things to draw:

```
# You CAN'T draw 10 samples without replacement from
#  a vector with length 5
sample(x = 1:5, size = 10)
```

Error: cannot take a sample larger than the population when 'replace = FALSE'

To fix this, just tell R that you want to sample with replacement:

```
# You CAN draw 10 samples with replacement from a
#  vector of length 5
sample(x = 1:5, size = 10, replace = TRUE)
##  [1] 4 2 2 3 3 4 5 3 2 5
```

To specify how likely each element in the vector `x` should be selected, use the `prob` argument. The length of the `prob` argument should be as long as the `x` argument. For example, let's draw 10 samples (with replacement) from the vector ["a", "b"], but we'll make the probability of selecting "a" to be .90, and the probability of selecting "b" to be .10

```
sample(x = c("a", "b"),
       prob = c(.9, .1),
       size = 10,
       replace = TRUE)
##  [1] "a" "a" "a" "a" "a" "a" "a" "a" "a" "a"
```

5.3.1.2 Ex: Simulating coin flips

Let's simulate 10 flips of a fair coin, were the probably of getting either a Head or Tail is .50. Because all values are equally likely, we don't need to specify the `prob` argument

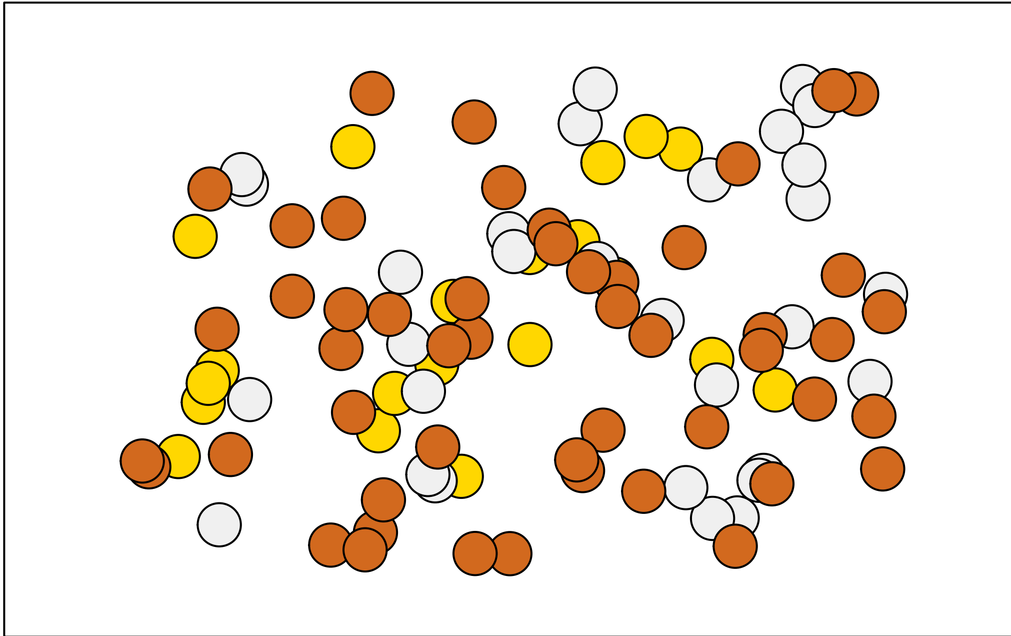
```
sample(x = c("H", "T"), # The possible values of the coin
       size = 10,  # 10 flips
       replace = TRUE) # Sampling with replacement
##  [1] "H" "H" "T" "T" "H" "T" "H" "T" "H" "H"
```

Now let's change it by simulating flips of a biased coin, where the probability of Heads is 0.8, and the probability of Tails is 0.2. Because the probabilities of each outcome are no longer equal, we'll need to specify them with the `prob` argument:

```
sample(x = c("H", "T"),
       prob = c(.8, .2), # Make the coin biased for Heads
       size = 10,
       replace = TRUE)
##  [1] "H" "H" "H" "H" "H" "T" "T" "H" "H" "H"
```

As you can see, our function returned a vector of 10 values corresponding to our sample size of 10.

5.3.1.3 Ex: Coins from a chest



Now, let's sample drawing coins from a treasure chest Let's say the chest has 100 coins: 20 gold, 30 silver, and 50 bronze. Let's draw 10 random coins from this chest.

```
# Create chest with the 100 coins

chest <- c(rep("gold", 20),
           rep("silver", 30),
           rep("bronze", 50))

# Draw 10 coins from the chest
sample(x = chest,
       size = 10)
##  [1] "bronze" "bronze" "bronze" "bronze" "bronze" "silver" "bronze"
##  [8] "bronze" "bronze" "silver"
```

The output of the `sample()` function above is a vector of 10 strings indicating the type of coin we drew on each sample. And like any random sampling function, this code will likely give you different results every time you run it! See how long it takes you to get 10 gold coins…

In the next section, we'll cover how to generate random data from specified *probability distributions*. What is a probability distribution? Well, it's simply an equation – also called a likelihood function – that indicates how likely certain numerical values are to be drawn.

We can use probability distributions to represent different types of data. For example, imagine you need to hire a new group of pirates for your crew. You have the option of hiring people from one of two different pirate training colleges that produce pirates of varying quality. One college "Pirate Training Unlimited" might tend to pirates that are generally ok - never great but never terrible. While another college "Unlimited Pirate Training" might produce pirates with a wide variety of quality, from very low to very high. In Figure 5.4 I plotted 5 example pirates from each college, where each pirate is shown as a ball with a number written on it. As you can see, pirates from PTU all tend to be clustered between 40 and 60 (not

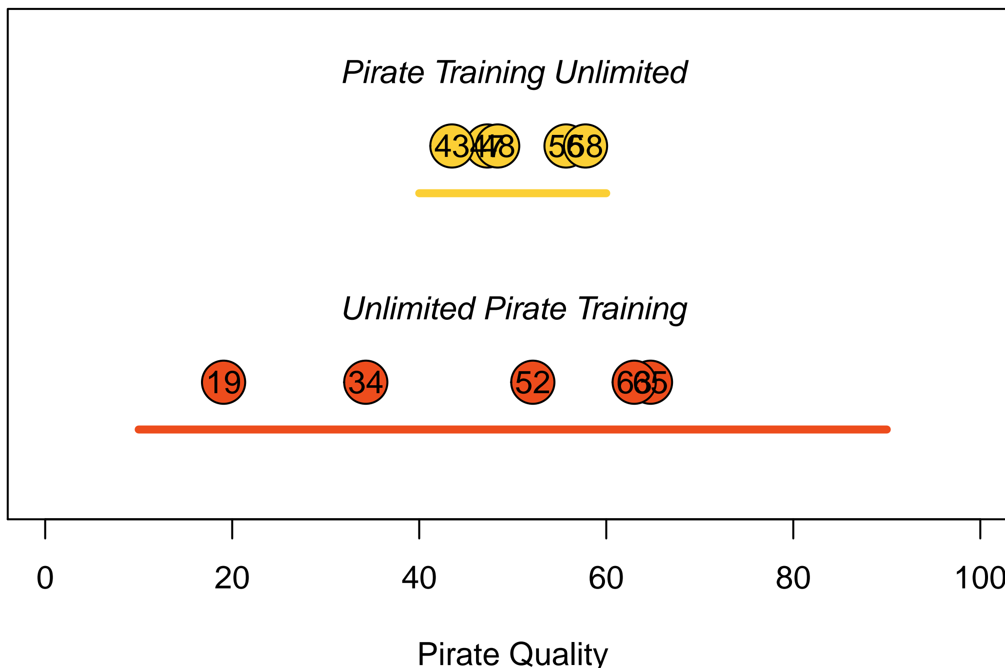


Figure 5.4: Sampling 5 potential pirates from two different pirate colleges. Pirate Training Unlimited (PTU) consistently produces average pirates (with scores between 40 and 60), while Unlimited Pirate Training (UPT), produces a wide range of pirates from 0 to 100.

terrible but not great), while pirates from UPT are all over the map, from 0 to 100. We can use probability distributions (in this case, the uniform distribution) to mathematically define how likely any possible value is to be drawn at random from a distribution. We could describe Pirate Training Unlimited with a uniform distribution with a small range, and Unlimited Pirate Training with a second uniform distribution with a wide range.

In the next two sections, I'll cover the two most common distributions: The Normal and the Uniform. However, R contains many more distributions than just these two. To see them all, look at the help menu for Distributions:

```
# See all distributions included in Base R
?Distributions
```

5.3.2 Normal (Gaussian)

Argument	Definition
`n`	The number of observations to draw from the distribution.
`mean`	The mean of the distribution.
`sd`	The standard deviation of the distribution.

The Normal (a.k.a "Gaussian") distribution is probably the most important distribution in all of statistics. The Normal distribution is bell-shaped, and has two parameters: a mean and a standard deviation. To generate samples from a normal distribution in R, we use the function `rnorm()`

```
# 5 samples from a Normal dist with mean = 0, sd = 1
rnorm(n = 5, mean = 0, sd = 1)
## [1] -1.38  2.40  0.37  0.53  0.69

# 3 samples from a Normal dist with mean = -10, sd = 15
rnorm(n = 3, mean = -10, sd = 15)
```

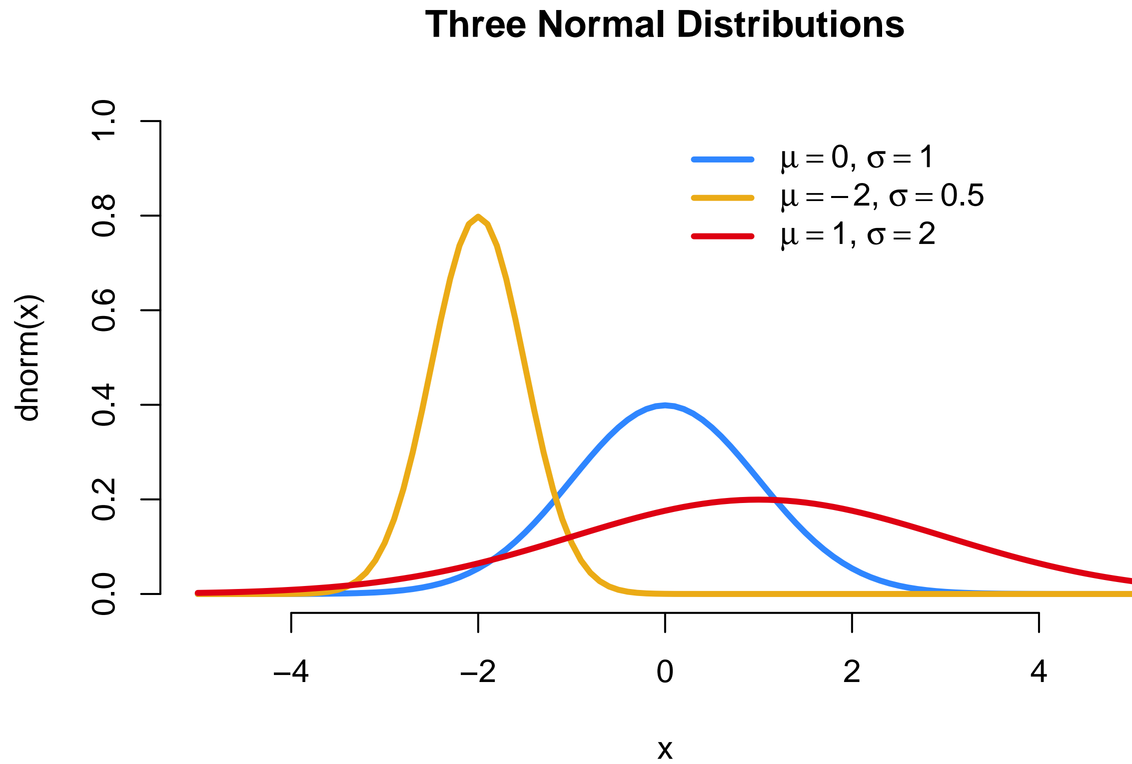


Figure 5.5: Three different normal distributions with different means and standard deviations

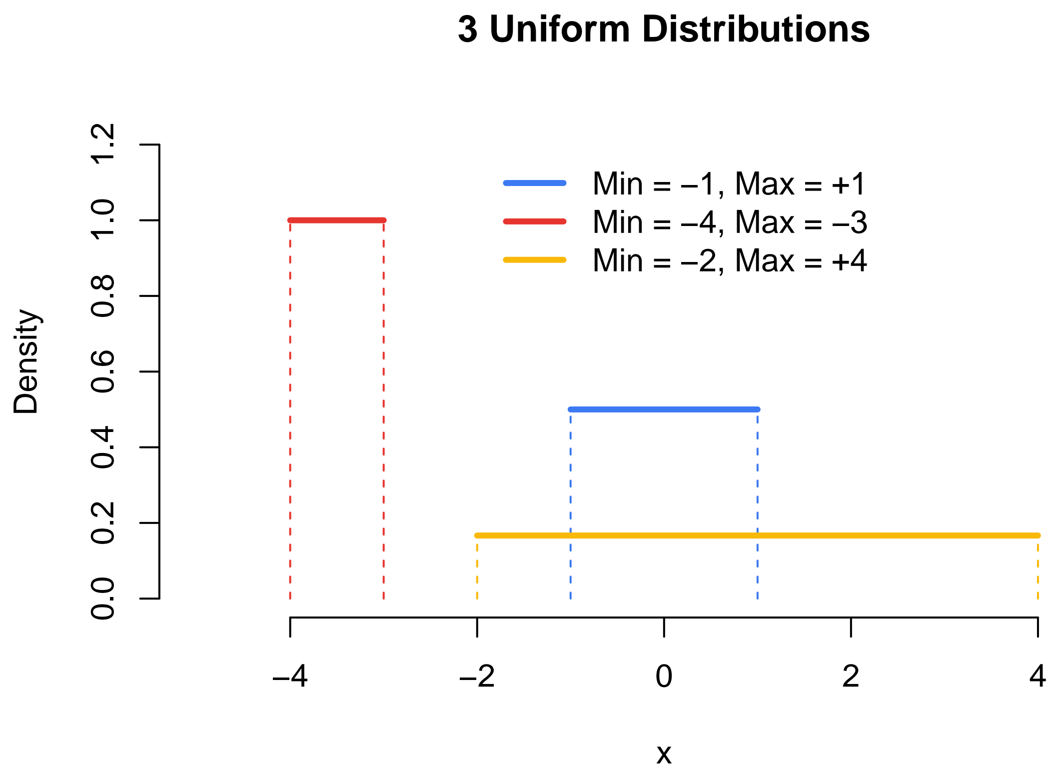


Figure 5.6: The Uniform distribution - known colloquially as the Anthony Davis distribution.

```
# 5 samples from Uniform dist with bounds at 0 and 1
runif(n = 5, min = 0, max = 1)
## [1] 0.94 0.80 0.57 0.11 0.22

# 10 samples from Uniform dist with bounds at -100 and +100
runif(n = 10, min = -100, max = 100)
##  [1]  10.94 -80.50  52.38   0.45 -79.65  31.94  87.28 -67.69 -58.64  -0.37
```

5.3.4 Notes on random samples

5.3.4.1 Random samples will always change

Every time you draw a sample from a probability distribution, you'll (likely) get a different result. For example, see what happens when I run the following two commands (you'll learn the `rnorm()` function on the next page…)

```
# Draw a sample of size 5 from a normal distribution with mean 100 and sd 10
rnorm(n = 5, mean = 100, sd = 10)
## [1] 107  94 103 106 100

# Do it again!
rnorm(n = 5, mean = 100, sd = 10)
## [1] 113 104 109 108  96
```

As you can see, the exact same code produced different results – and that's exactly what we want! Each time you run `rnorm()`, or another distribution function, you'll get a new random sample.

5.3.4.2 Use `set.seed()` to control random samples

There will be cases where you will want to exert some control over the random samples that R produces from sampling functions. For example, you may want to create a reproducible example of some code that anyone else can replicate exactly. To do this, use the `set.seed()` function. Using `set.seed()` will force R to produce consistent random samples at any time on any computer.

In the code below I'll set the sampling seed to 100 with `set.seed(100)`. I'll then run `rnorm()` twice. The results will always be consistent (because we fixed the sampling seed).

```
# Fix sampling seed to 100, so the next sampling functions
#   always produce the same values
set.seed(100)

# The result will always be -0.5022, 0.1315, -0.0789
rnorm(3, mean = 0, sd = 1)
## [1] -0.502  0.132 -0.079

# The result will always be 0.887, 0.117, 0.319
rnorm(3, mean = 0, sd = 1)
## [1] 0.89 0.12 0.32
```

Try running the same code on your machine and you'll see the exact same samples that I got above. Oh and the value of 100 I used above in `set.seed(100)` is totally arbitrary – you can set the seed to any integer you want. I just happen to like how `set.seed(100)` looks in my code.

5.4 Test your R might!

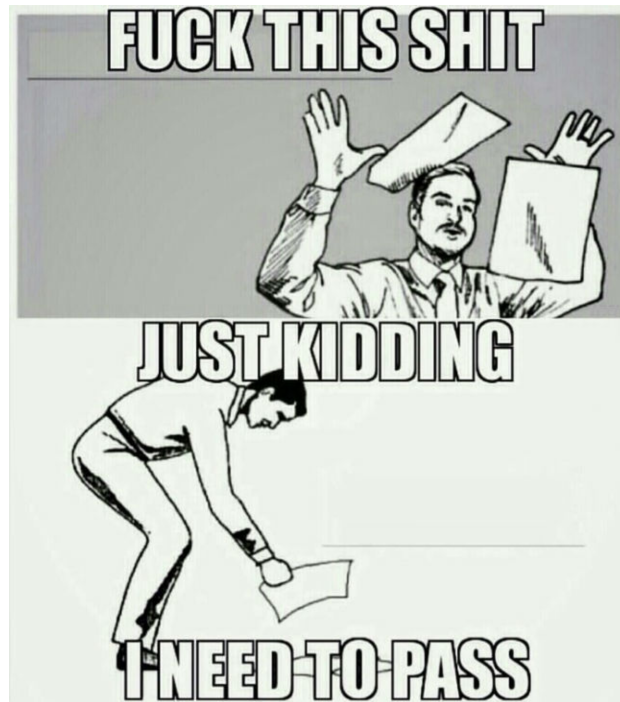
1. Create the vector [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] in three ways: once using `c()`, once using `a:b`, and once using `seq()`.
2. Create the vector [2.1, 4.1, 6.1, 8.1] in two ways, once using `c()` and once using `seq()`
3. Create the vector [0, 5, 10, 15] in 3 ways: using `c()`, `seq()` with a `by` argument, and `seq()` with a `length.out` argument.
4. Create the vector [101, 102, 103, 200, 205, 210, 1000, 1100, 1200] using a combination of the `c()` and `seq()` functions
5. A new batch of 100 pirates are boarding your ship and need new swords. You have 10 scimitars, 40 broadswords, and 50 cutlasses that you need to distribute evenly to the 100 pirates as they board. Create a vector of length 100 where there is 1 scimitar, 4 broadswords, and 5 cutlasses in each group of 10. That is, in the first 10 elements there should be exactly 1 scimitar, 4 broadswords and 5 cutlasses. The next 10 elements should also have the same number of each sword (and so on).
6. Create a vector that repeats the integers from 1 to 5, 10 times. That is [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, …]. The length of the vector should be 50!
7. Now, create the same vector as before, but this time repeat 1, 10 times, then 2, 10 times, etc., That is [1, 1, 1, …, 2, 2, 2, …, … 5, 5, 5]. The length of the vector should also be 50
8. Create a vector containing 50 samples from a Normal distribution with a population mean of 20 and standard deviation of 2.
9. Create a vector containing 25 samples from a Uniform distribution with a lower bound of -100 and an upper bound of -50.

Chapter 6

Vector functions

In this chapter, we'll cover the core functions for vector objects. The code below uses the functions you'll learn to calculate summary statistics from two exams.

```
# 10 students from two different classes took two exams.
#  Here are three vectors showing the data
midterm <- c(62, 68, 75, 79, 55, 62, 89, 76, 45, 67)
final <- c(78, 72, 97, 82, 60, 83, 92, 73, 50, 88)

# How many students are there?
length(midterm)
## [1] 10

# Add 5 to each midterm score (extra credit!)
midterm <- midterm + 5
midterm
##  [1] 67 73 80 84 60 67 94 81 50 72

# Difference between final and midterm scores
final - midterm
##  [1] 11 -1 17 -2  0 16 -2 -8  0 16

# Each student's average score
(midterm + final) / 2
##  [1] 72 72 88 83 60 75 93 77 50 80

# Mean midterm grade
mean(midterm)
## [1] 73

# Standard deviation of midterm grades
sd(midterm)
## [1] 13

# Highest final grade
max(final)
## [1] 97

# z-scores
```




```
midterm.z <- (midterm - mean(midterm)) / sd(midterm)
final.z <- (final - mean(final)) / sd(final)
```

6.1 Arithmetic operations on vectors

So far, you know how to do basic arithmetic operations like + (addition), - (subtraction), and * (multiplication) on scalars. Thankfully, R makes it just as easy to do arithmetic operations on numeric vectors:

```
a <- c(1, 2, 3, 4, 5)
b <- c(10, 20, 30, 40, 50)

a + 100
## [1] 101 102 103 104 105
a + b
## [1] 11 22 33 44 55
(a + b) / 10
## [1] 1.1 2.2 3.3 4.4 5.5
```

If you do an operation on a vector with a scalar, R will apply the scalar to each element in the vector. For example, if you have a vector and want to add 10 to each element in the vector, just add the vector and scalar objects. Let's create a vector with the integers from 1 to 10, and add then add 100 to each element:

```
# Take the integers from 1 to 10, then add 100 to each
1:10 + 100
##  [1] 101 102 103 104 105 106 107 108 109 110
```

As you can see, the result is [1 + 100, 2 + 100, ... 10 + 100]. Of course, we could have made this vector with the `a:b` function like this: `101:110`, but you get the idea.

Of course, this doesn't only work with addition…oh no. Let's try division, multiplication, and exponents. Let's create a vector **a** with the integers from 1 to 10 and then change it up:

```
a <- 1:10
a / 100
##  [1] 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.10
a ^ 2
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Again, if you perform an algebraic operation on a vector with a scalar, R will just apply the operation to every element in the vector.

6.1.1 Basic math with multiple vectors

What if you want to do some operation on two vectors of the same length? Easy. Just apply the operation to both vectors. R will then combine them element–by–element. For example, if you add the vector [1, 2, 3, 4, 5] to the vector [5, 4, 3, 2, 1], the resulting vector will have the values [1 + 5, 2 + 4, 3 + 3, 4 + 2, 5 + 1] = [6, 6, 6, 6, 6]:

```
c(1, 2, 3, 4, 5) + c(5, 4, 3, 2, 1)
## [1] 6 6 6 6 6
```

Let's create two vectors a and b where each vector contains the integers from 1 to 5. We'll then create two new vectors **ab.sum**, the sum of the two vectors and **ab.diff**, the difference of the two vectors, and **ab.prod**, the product of the two vectors:

```
a <- 1:5
b <- 1:5

ab.sum <- a + b
ab.diff <- a - b
ab.prod <- a * b

ab.sum
## [1]  2  4  6  8 10
ab.diff
## [1] 0 0 0 0 0
ab.prod
## [1]  1  4  9 16 25
```

6.1.2 Ex: Pirate Bake Sale

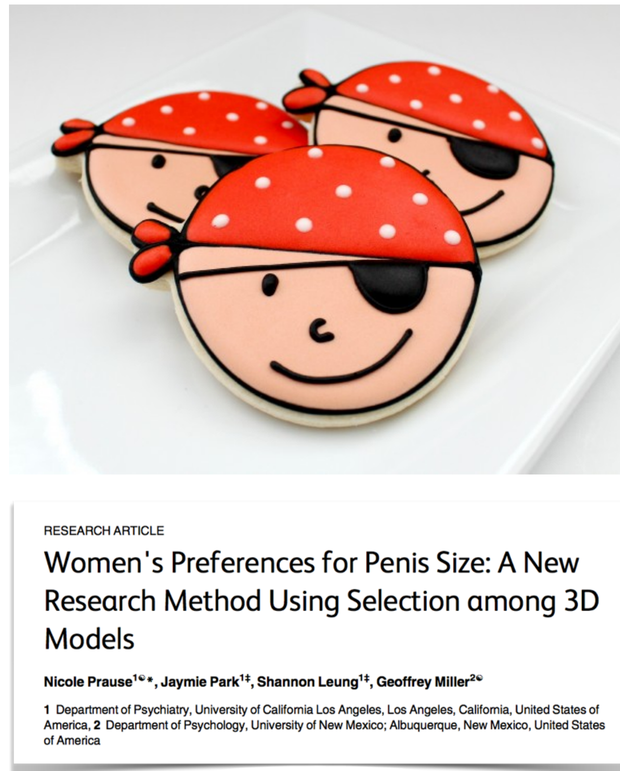
Let's say you had a bake sale on your ship where 5 pirates sold both pies and cookies. You could record the total number of pies and cookies sold in two vectors:

```
pies <- c(3, 6, 2, 10, 4)
cookies <- c(70, 40, 40, 200, 60)
```

Now, let's say you want to know how many total items each pirate sold. You can do this by just adding the two vectors:

```
total.sold <- pies + cookies
total.sold
## [1]  73  46  42 210  64
```

Crazy.



RESEARCH ARTICLE

Women's Preferences for Penis Size: A New Research Method Using Selection among 3D Models

Nicole Prause[1]*, Jaymie Park[1‡], Shannon Leung[1‡], Geoffrey Miller[2]

1 Department of Psychiatry, University of California Los Angeles, Los Angeles, California, United States of America, 2 Department of Psychology, University of New Mexico; Albuquerque, New Mexico, United States of America

Figure 6.1: According to this article published in 2015 in Plos One, when it comes to people, length may matter for some. But trust me, for vectors it always does.

6.2 Summary statistics

Ok, now that we can create vectors, let's learn the basic descriptive statistics functions. We'll start with functions that apply to continuous data. Continuous data is data that, generally speaking, can take on an infinite number of values. Height and weight are good examples of continuous data. Table 6.1 contains common functions for continuous, numeric vectors. Each of them takes a numeric vector as an argument, and returns either a scalar (or in the case of `summary()`, a `table`) as a result.

Table 6.1: Summary statistic functions for continuous data.

Function	Example	Result
`sum(x), product(x)`	`sum(1:10)`	55
`min(x), max(x)`	`min(1:10)`	1
`mean(x), median(x)`	`mean(1:10)`	5.5
`sd(x), var(x), range(x)`	`sd(1:10)`	3.03
`quantile(x, probs)`	`quantile(1:10, probs = .2)`	2.8
`summary(x)`	`summary(1:10)`	`Min = 1.00. 1st Qu. = 3.25, Median = 5.50, Mean = 5.50, 3rd Qu. = 7.75, Max = 10.0`

Let's calculate some descriptive statistics from some pirate related data. I'll create a vector called `x` that contains the number of tattoos from 10 random pirates.

```
tattoos <- c(4, 50, 2, 39, 4, 20, 4, 8, 10, 100)
```

Now, we can calculate several descriptive statistics on this vector by using the summary statistics functions:

```
min(tattoos)
## [1] 2
mean(tattoos)
## [1] 24
```

sometimes...). Instead, use `length()` function. The `length()` function takes a vector as an argument, and returns a scalar representing the number of elements in the vector:

```
a <- 1:10
length(a)  # How many elements are in a?
## [1] 10

b <- seq(from = 1, to = 100, length.out = 20)
length(b)  # How many elements are in b?
## [1] 20

length(c("This", "character", "vector", "has", "six", "elements."))
## [1] 6
length("This character scalar has just one element.")
## [1] 1
```

Get used to the `length()` function people, you'll be using it a lot!

6.2.2 Additional numeric vector functions

Table 6.2 contains additional functions that you will find useful when managing numeric vectors:

Table 6.2: Vector summary functions for continuous data.

Function	Description	Example	Result
`round(x, digits)`	Round elements in x to `digits` digits	`round(c(2.231, 3.1415), digits = 1)`	2.2, 3.1
`ceiling(x), floor(x)`	Round elements x to the next highest (or lowest) integer	`ceiling(c(5.1, 7.9))`	6, 8
`x %% y`	Modular arithmetic (ie. x mod y)	`7 %% 3`	1

6.2.3 Sample statistics from random samples

Now that you know how to calculate summary statistics, let's take a closer look at how R draws random samples using the `rnorm()` and `runif()` functions. In the next code chunk, I'll calculate some summary statistics from a vector of 5 values from a Normal distribution with a mean of 10 and a standard deviation of 5. I'll then calculate summary statistics from this sample using `mean()` and `sd()`:

```
# 5 samples from a Normal dist with mean = 10 and sd = 5
x <- rnorm(n = 5, mean = 10, sd = 5)

# What are the mean and standard deviation of the sample?
mean(x)
## [1] 11
sd(x)
## [1] 2.5
```

As you can see, the mean and standard deviation of our sample vector are close to the population values of 10 and 5 – but they aren't exactly the same because these are sample data. If we take a much larger sample (say, 100,000), the sample statistics should get much closer to the population values:

```
# 100,000 samples from a Normal dist with mean = 10, sd = 5
y <- rnorm(n = 100000, mean = 10, sd = 5)

mean(y)
## [1] 10
sd(y)
## [1] 5
```

Yep, sure enough our new sample y (containing 100,000 values) has a sample mean and standard deviation much closer (almost identical) to the population values than our sample x (containing only 5 values). This is an example of what is called the law of large numbers. Google it.

6.3 Counting statistics

Next, we'll move on to common counting functions for vectors with discrete or non-numeric data. Discrete data are those like gender, occupation, and monkey farts, that only allow for a finite (or at least, plausibly finite) set of responses. Common functions for discrete vectors are in Table 6.3. Each of these vectors takes a vector as an argument – however, unlike the previous functions we looked at, the arguments to these functions can be either numeric or character.

Table 6.3: Counting functions for discrete data.

Function	Description	Example	Result
`unique(x)`	Returns a vector of all unique values.	`unique(c(1, 1, 2, 10))`	1, 2, 10
`table(x, exclude)`	Returns a table showing all the unique values as well as a count of each occurrence. To include a count of NA values, include the argument `exclude = NULL`	`table(c("a", "a", "b", "c"))`	`2-"a", 1-"b", 1-"c"`

Let's test these functions by starting with two vectors of discrete data:

```
vec <- c(1, 1, 1, 5, 1, 1, 10, 10, 10)
gender <- c("M", "M", "F", "F", "F", "M", "F", "M", "F")
```

The function `unique(x)` will tell you all the unique values in the vector, but won't tell you anything about how often each value occurs.

```
unique(vec)
## [1]  1  5 10
unique(gender)
## [1] "M" "F"
```

The function `table()` does the same thing as `unique()`, but goes a step further in telling you how often each of the unique values occurs:

```
table(vec)
## vec
##  1  5 10
##  5  1  3
```

```
table(gender)
## gender
## F M
## 5 4
```

If you want to get a table of percentages instead of counts, you can just divide the result of the `table()` function by the sum of the result:

```
table(vec) / sum(table(vec))
## vec
##    1    5   10
## 0.56 0.11 0.33
table(gender) / sum(table(gender))
## gender
##    F    M
## 0.56 0.44
```

6.4 Missing (NA) values

In R, missing data are coded as NA. In real datasets, NA values turn up all the time. Unfortunately, most descriptive statistics functions will freak out if there is a missing (NA) value in the data. For example, the following code will return NA as a result because there is an NA value in the data vector:

```
a <- c(1, 5, NA, 2, 10)
mean(a)
## [1] NA
```

Thankfully, there's a way we can work around this. To tell a descriptive statistic function to ignore missing (NA) values, include the argument `na.rm = TRUE` in the function. This argument explicitly tells the function to ignore NA values. Let's try calculating the mean of the vector `a` again, this time with the additional`na.rm = TRUE` argument:

```
mean(a, na.rm = TRUE)
## [1] 4.5
```

Now, the function ignored the NA value and returned the mean of the remaining data. While this may seem trivial now (why did we include an NA value in the vector if we wanted to ignore it?!), it will be become very important when we apply the function to real data which, very often, contains missing values.

6.5 Standardization (z-score)

A common task in statistics is to standardize variables – also known as calculating z-scores. The purpose of standardizing a vector is to put it on a common scale which allows you to compare it to other (standardized) variables. To standardize a vector, you simply subtract the vector by its mean, and then divide the result by the vector's standard deviation.

If the concept of z-scores is new to you – don't worry. In the next worked example, you'll see how it can help you compare two sets of data. But for now, let's see how easy it is to standardize a vector using basic arithmetic.

Let's say you have a vector a containing some data. We'll assign the vector to a new object called `a` then calculate the mean and standard deviation with the `mean()` and `sd()` functions:

Table 6.4: Scores from a pirate competition

pirate	grogg	climbing
Heidi	12	100
Andrew	8	520
Becki	1	430
Madisen	6	200
David	2	700

```
a <- c(5, 3, 7, 5, 5, 3, 4)
mean(a)
## [1] 4.6
sd(a)
## [1] 1.4
```

Ok. Now we'll create a new vector called `a.z` which is a standardized version of a. To do this, we'll simply subtract the mean of the vector, then divide by the standard deviation.

```
a.z <- (a - mean(a)) / sd(a)
```

Now let's look at the standardized values:

```
a.z
## [1]  0.31 -1.12  1.74  0.31  0.31 -1.12 -0.41
```

The mean of `a.z` should now be 0, and the standard deviation of `a.z` should now be 1. Let's make sure:

```
mean(a.z)
## [1] 2e-16
sd(a.z)
## [1] 1
```

Sweet. Oh, don't worry that the mean of `a.z` doesn't look like exactly zero. Using non-scientific notation, the result is 0.000000000000000198. For all intents and purposes, that's 0. The reason the result is not exactly 0 is due to computer science theoretical reasons that I cannot explain (because I don't understand them).

6.5.1 Ex: Evaluating a competition

Your gluten-intolerant first mate just perished in a tragic soy sauce incident and it's time to promote another member of your crew to the newly vacated position. Of course, only two qualities really matter for a pirate: rope-climbing, and grogg drinking. Therefore, to see which of your crew deserves the promotion, you decide to hold a climbing and drinking competition. In the climbing competition, you measure how many feet of rope a pirate can climb in an hour. In the drinking competition, you measure how many mugs of grogg they can drink in a minute. Five pirates volunteer for the competition – here are their results:

We can represent the main results with two vectors `grogg` and `climbing`:

```
grogg <- c(12, 8, 1, 6, 2)
climbing <- c(100, 520, 430, 200, 700)
```

Now you've got the data, but there's a problem: the scales of the numbers are very different. While the grogg numbers range from 1 to 12, the climbing numbers have a much larger range from 100 to 700. This makes it difficult to compare the two sets of numbers directly.

Table 6.5: Renata's treasure haul when she was sober and when she was drunk

day	sober	drunk
Monday	2	0
Tuesday	0	0
Wednesday	3	1
Thursday	1	0
Friday	0	1
Saturday	3	2
Sunday	5	2

To solve this problem, we'll use standardization. Let's create new standardized vectors called `grogg.z` and `climbing.z`

```
grogg.z <- (grogg - mean(grogg)) / sd(grogg)
climbing.z <- (climbing - mean(climbing)) / sd(climbing)
```

Now let's look at the final results

```
grogg.z
## [1]  1.379  0.489 -1.068  0.044 -0.845
climbing.z
## [1] -1.20  0.54  0.17 -0.78  1.28
```

It looks like there were two outstanding performances in particular. In the grogg drinking competition, the first pirate (Heidi) had a z-score of 1.4. We can interpret this by saying that Heidi drank 1.4 more standard deviations of mugs of grogg than the average pirate. In the climbing competition, the fifth pirate (David) had a z-score of 1.3. Here, we would conclude that David climbed 1.3 standard deviations more than the average pirate.

But which pirate was the best on average across both events? To answer this, let's create a combined z-score for each pirate which calculates the average z-scores for each pirate across the two events. We'll do this by adding two performances and dividing by two. This will tell us, how good, on average, each pirate did relative to her fellow pirates.

```
average.z <- (grogg.z + (climbing.z)) / 2
```

Let's look at the result:

```
round(average.z, 1)
## [1]  0.1  0.5 -0.5 -0.4  0.2
```

The highest average z-score belongs to the second pirate (Andrew) who had an average z-score value of 0.5. The first and last pirates, who did well in one event, seemed to have done poorly in the other event.

Moral of the story: promote the pirate who can drink *and* climb.

6.6 Test your R Might!

1. Create a vector that shows the square root of the integers from 1 to 10.
2. Renata thinks that she finds more treasure when she's had a mug of grogg than when she doesn't. To test this, she recorded how much treasure she found over 7 days without drinking any grogg (ie., sober), and then did the same over 7 days while drinking grogg (ie., drunk). Here are her results:

How much treasure did Renata find on average when she was sober? What about when she was drunk?

3. Using Renata's data again, create a new vector called `difference` that shows how much more treasure Renata found when she was drunk and when she was not. What was the mean, median, and standard deviation of the difference?

4. There's an old parable that goes something like this. A man does some work for a king and needs to be paid. Because the man loves rice (who doesn't?!), the man offers the king two different ways that he can be paid. *You can either pay me 100 kilograms of rice, or, you can pay me as follows: get a chessboard and put one grain of rice in the top left square. Then put 2 grains of rice on the next square, followed by 4 grains on the next, 8 grains on the next...and so on, where the amount of rice doubles on each square, until you get to the last square. When you are finished, give me all the grains of rice that would (in theory), fit on the chessboard.* The king, sensing that the man was an idiot for making such a stupid offer, immediately accepts the second option. He summons a chessboard, and begins counting out grains of rice one by one... Assuming that there are 64 squares on a chessboard, calculate how many grains of rice the main will receive. If one grain of rice weights 1/6400 kilograms, how many kilograms of rice did he get? *Hint: If you have trouble coming up with the answer, imagine how many grains are on the first, second, third and fourth squares, then try to create the vector that shows the number of grains on each square. Once you come up with that vector, you can easily calculate the final answer with the `sum()` function.*

Chapter 7

Indexing Vectors with [ ]

boat.names	boat.colors	boat.ages	boat.prices	boat.costs
a	black	143	53	52
b	green	53	87	80
c	pink	356	54	20
d	blue	23	66	100
e	blue	647	264	189
f	green	24	32	12
g	green	532	532	520
h	yellow	43	58	68
i	black	66	99	80
j	black	86	132	100

```
# Boat sale. Creating the data vectors
boat.names <- c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j")
boat.colors <- c("black", "green", "pink", "blue", "blue",
                "green", "green", "yellow", "black", "black")
boat.ages <- c(143, 53, 356, 23, 647, 24, 532, 43, 66, 86)
boat.prices <- c(53, 87, 54, 66, 264, 32, 532, 58, 99, 132)
boat.costs <- c(52, 80, 20, 100, 189, 12, 520, 68, 80, 100)

# What was the price of the first boat?
boat.prices[1]
## [1] 53

# What were the ages of the first 5 boats?
boat.ages[1:5]
## [1] 143  53 356  23 647

# What were the names of the black boats?
boat.names[boat.colors == "black"]
## [1] "a" "i" "j"

# What were the prices of either green or yellow boats?
boat.prices[boat.colors == "green" | boat.colors == "yellow"]
## [1]  87  32 532  58

# Change the price of boat "s" to 100
boat.prices[boat.names == "s"] <- 100
```

```
# What was the median price of black boats less than 100 years old?
median(boat.prices[boat.colors == "black" & boat.ages < 100])
## [1] 116

# How many pink boats were there?
sum(boat.colors == "pink")
## [1] 1

# What percent of boats were older than 100 years old?
mean(boat.ages < 100)
## [1] 0.6
```

By now you should be a whiz at applying functions like `mean()` and `table()` to vectors. However, in many analyses, you won't want to calculate statistics of an entire vector. Instead, you will want to access specific *subsets* of values of a vector based on some criteria. For example, you may want to access values in a specific location in the vector (i.e.; the first 10 elements) or based on some criteria within that vector (i.e.; all values greater than 0), or based on criterion from values in a *different* vector (e.g.; All values of age where sex is Female). To access specific values of a vector in R, we use *indexing* using brackets `[]`. In general, whatever you put inside the brackets, tells R which values of the vector object you want. There are two main ways that you can use indexing to access subsets of data in a vector: numerical and logical indexing.

7.1 Numerical Indexing

With numerical indexing, you enter a vector of integers corresponding to the values in the vector you want to access in the form `a[index]`, where `a` is the vector, and `index` is a vector of index values. For example, let's use numerical indexing to get values from our boat vectors.

```
# What is the first boat name?
boat.names[1]
## [1] "a"

# What are the first five boat colors?
boat.colors[1:5]
## [1] "black" "green" "pink"  "blue"  "blue"

# What is every second boat age?
boat.ages[seq(1, 5, by = 2)]
## [1] 143 356 647
```

You can use any indexing vector as long as it contains integers. You can even access the same elements multiple times:

```
# What is the first boat age (3 times)
boat.ages[c(1, 1, 1)]
## [1] 143 143 143
```

If it makes your code clearer, you can define an indexing object before doing your actual indexing. For example, let's define an object called `my.index` and use this object to index our data vector:

```
my.index <- 3:5
boat.names[my.index]
## [1] "c" "d" "e"
```

Figure 7.1: Logical indexing. Good for R aliens and R pirates.

7.2 Logical Indexing

The second way to index vectors is with *logical vectors*. A logical vector is a vector that *only* contains TRUE and FALSE values. In R, true values are designated with TRUE, and false values with FALSE. When you index a vector with a logical vector, R will return values of the vector for which the indexing vector is TRUE. If that was confusing, think about it this way: a logical vector, combined with the brackets `[ ]`, acts as a *filter* for the vector it is indexing. It only lets values of the vector pass through for which the logical vector is TRUE.

You could create logical vectors directly using `c()`. For example, I could access every other value of the following vector as follows:

```
a <- c(1, 2, 3, 4, 5)
a[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
## [1] 1 3 5
```

As you can see, R returns all values of the vector `a` for which the logical vector is TRUE.

However, creating logical vectors using `c()` is tedious. Instead, it's better to create logical vectors from *existing vectors* using comparison operators like < (less than), == (equals to), and != (not equal to). A complete list of the most common comparison operators is in Figure 7.3. For example, let's create some logical vectors from our `boat.ages` vector:

```
# Which ages are > 100?
boat.ages > 100
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE

# Which ages are equal to 23?
boat.ages == 23
##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE

# Which boat names are equal to c?
boat.names == "c"
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

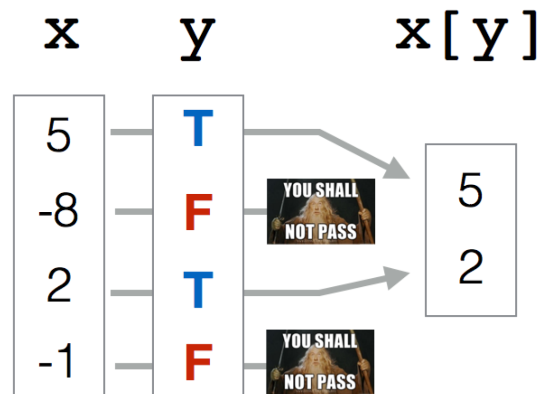


Figure 7.2: FALSE values in a logical vector are like lots of mini-Gandolfs. In this example, I am indexing a vector x with a logical vector y (y for example could be x > 0, so all positive values of x are TRUE and all negative values are FALSE). The result is a vector of length 2, which are the values of x for which the logical vector y was true. Gandolf stopped all the values of x for which y was FALSE.

```
==      equal
!=      not equal
<       less than
<=      less than or equal
>       greater than
>=      greater than or equal
|       or
!       not
%in%    in the set
```

Figure 7.3: Logical comparison operators in R

You can also create logical vectors by comparing a vector to another vector of the same length. When you do this, R will compare values in the same position (e.g.; the first values will be compared, then the second values, etc.). For example, we can compare the `boat.cost` and `boat.price` vectors to see which boats sold for a higher price than their cost:

```
# Which boats had a higher price than cost?
boat.prices > boat.costs
##  [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE

# Which boats had a lower price than cost?
boat.prices < boat.costs
##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE
```

Once you've created a logical vector using a comparison operator, you can use it to index any vector with the same length. Here, I'll use logical vectors to get the prices of boats whose ages were greater than 100:

```
# What were the prices of boats older than 100?
boat.prices[boat.ages > 100]
## [1]  53  54 264 532
```

Here's how logical indexing works step-by-step:

```
# Which boats are older than 100 years?
boat.ages > 100
##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE

# Writing the logical index by hand (you'd never do this!)
#  Show me all of the boat prices where the logical vector is TRUE:
boat.prices[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, FALSE, FALSE)]
## [1]  53  54 264 532

# Doing it all in one step! You get the same answer:
boat.prices[boat.ages > 100]
## [1]  53  54 264 532
```

7.2.1 `&` (and), `|` (or), `%in%`

In addition to using single comparison operators, you can combine multiple logical vectors using the OR (which looks like `|` and AND `&` commands. The OR `|` operation will return TRUE if any of the logical vectors is TRUE, while the AND `&` operation will only return TRUE if all of the values in the logical vectors is TRUE. This is especially powerful when you want to create a logical vector based on criteria from multiple vectors.

For example, let's create a logical vector indicating which boats had a price greater than 200 OR less than 100, and then use that vector to see what the names of these boats were:

```
# Which boats had prices greater than 200 OR less than 100?
boat.prices > 200 | boat.prices < 100
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE

# What were the NAMES of these boats
boat.names[boat.prices > 200 | boat.prices < 100]
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

You can combine as many logical vectors as you want (as long as they all have the same length!):

```
# Boat names of boats with a color of black OR with a price > 100
boat.names[boat.colors == "black" | boat.prices > 100]
## [1] "a" "e" "g" "i" "j"

# Names of blue boats with a price greater than 200
boat.names[boat.colors == "blue" & boat.prices > 200]
## [1] "e"
```

You can combine as many logical vectors as you want to create increasingly complex selection criteria. For example, the following logical vector returns TRUE for cases where the boat colors are black OR brown, AND where the price was less than 100:

```
# Which boats were eithe black or brown, AND had a price less than 100?
(boat.colors == "black" | boat.colors == "brown") & boat.prices < 100
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE

# What were the names of these boats?
boat.names[(boat.colors == "black" | boat.colors == "brown") & boat.prices < 100]
## [1] "a" "i"
```

When using multiple criteria, make sure to use parentheses when appropriate. If I didn't use parentheses above, I would get a different answer.

The `%in%` operation helps you to easily create multiple OR arguments.Imagine you have a vector of categorical data that can take on many different values. For example, you could have a vector x indicating people's favorite letters.

```
x <- c("a", "t", "a", "b", "z")
```

Now, let's say you want to create a logical vector indicating which values are either a or b or c or d. You could create this logical vector with multiple | (OR) commands:

```
x == "a" | x == "b" | x == "c" | x == "d"
## [1]  TRUE FALSE  TRUE  TRUE FALSE
```

However, this takes a long time to write. Thankfully, the `%in%` operation allows you to combine multiple OR comparisons much faster. To use the `%in%` function, just put it in between the original vector, and a new vector of possible values. The `%in%` function goes through every value in the vector x, and returns TRUE if it finds it in the vector of possible values – otherwise it returns FALSE.

```
x %in% c("a", "b", "c", "d")
## [1]  TRUE FALSE  TRUE  TRUE FALSE
```

As you can see, the result is identical to our previous result.

7.2.2 Counts and percentages from logical vectors

Many (if not all) R functions will interpret TRUE values as 1 and FALSE values as 0. This allows us to easily answer questions like "How many values in a data vector are greater than 0?" or "What percentage of values are equal to 5?" by applying the `sum()` or `mean()` function to a logical vector.

We'll start with a vector x of length 10, containing 3 positive numbers and 5 negative numbers.

```
x <- c(1, 2, 3, -5, -5, -5, -5, -5)
```

We can create a logical vector to see which values are greater than 0:

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

Now, we'll use `sum()` and `mean()` on that logical vector to see how many of the values in x are positive, and what percent are positive. We should find that there are 5 TRUE values, and that 50% of the values (5 / 10) are TRUE.

```
sum(x > 0)
## [1] 3
mean(x > 0)
## [1] 0.38
```

This is a *really* powerful tool. Pretty much *any* time you want to answer a question like "How many of X are Y" or "What percent of X are Y", you use `sum()` or `mean()` function with a logical vector as an argument.

7.2.3 Additional Logical functions

R has lots of special functions that take vectors as arguments, and return logical vectors based on multiple criteria. For example, you can use the `is.na()` function to test which values of a vector are missing. Table 7.1 contains some that I frequently use:

Table 7.1: Functions to create and use logical vectors.

Function	Description	Example	Result
`is.na(x)`	Which values in x are NA?	`is.na(c(2, NA, 5))`	FALSE, TRUE, FALSE
`is.finite(x)`	Which values in x are numbers?	`is.finite(c(NA, 89, 0))`	FALSE, TRUE, TRUE
`duplicated(x)`	Which values in x are duplicated?	`duplicated(c(1, 4, 1, 2))`	FALSE, FALSE, TRUE, FALSE
`which(x)`	Which values in x are TRUE?	`which(c(TRUE, FALSE, TRUE))`	1, 3

Logical vectors aren't just good for indexing, you can also use them to figure out which values in a vector satisfy some criteria. To do this, use the function `which()`. If you apply the function `which()` to a logical vector, R will tell you which values of the index are TRUE. For example:

```
# A vector of sex information
sex <- c("m", "m", "f", "m", "f", "f")

# Which values of sex are m?
which(sex == "m")
## [1] 1 2 4

# Which values of sex are f?
which(sex == "f")
## [1] 3 5 6
```

7.3 Changing values of a vector

Now that you know how to index a vector, you can easily change specific values in a vector using the assignment (`<-`) operation. To do this, just assign a vector of new values to the indexed values of the original vector:

Let's create a vector `a` which contains 10 1s:

```
a <- rep(1, 10)
```

Now, let's change the first 5 values in the vector to 9s by indexing the first five values, and assigning the value of 9:

```
a[1:5] <- 9
a
##  [1] 9 9 9 9 9 1 1 1 1 1
```

Now let's change the last 5 values to 0s. We'll index the values 6 through 10, and assign a value of 0.

```
a[6:10] <- 0
a
##  [1] 9 9 9 9 9 0 0 0 0 0
```

Of course, you can also change values of a vector using a logical indexing vector. For example, let's say you have a vector of numbers that should be from 1 to 10. If values are outside of this range, you want to set them to either the minimum (1) or maximum (10) value:

```
# x is a vector of numbers that should be from 1 to 10
x <- c(5, -5, 7, 4, 11, 5, -2)

# Assign values less than 1 to 1
x[x < 1] <- 1

# Assign values greater than 10 to 10
x[x > 10] <- 10

# Print the result!
x
## [1]  5  1  7  4 10  5  1
```

As you can see, our new values of x are now never less than 1 or greater than 10!

A note on indexing...

Technically, when you assign new values to a vector, you should always assign a vector of the same length as the number of values that you are updating. For example, given a vector a with 10 1s:

```
a <- rep(1, 10)
```

To update the first 5 values with 5 9s, we should assign a new vector of 5 9s

```
a[1:5] <- c(9, 9, 9, 9, 9)
a
##  [1] 9 9 9 9 9 1 1 1 1 1
```

However, if we repeat this code but just assign a single 9, R will repeat the value as many times as necessary to fill the indexed value of the vector. That's why the following code still works:

```
a[1:5] <- 9
a
```

```
##  [1] 9 9 9 9 9 1 1 1 1 1
```

In other languages this code wouldn't work because we're trying to replace 5 values with just 1. However, this is a case where R bends the rules a bit.

7.3.1 Ex: Fixing invalid responses to a Happiness survey

Assigning and indexing is a particularly helpful tool when, for example, you want to remove invalid values in a vector before performing an analysis. For example, let's say you asked 10 people how happy they were on a scale of 1 to 5 and received the following responses:

```
happy <- c(1, 4, 2, 999, 2, 3, -2, 3, 2, 999)
```

As you can see, we have some invalid values (999 and -2) in this vector. To remove them, we'll use logical indexing to change the invalid values (999 and -2) to NA. We'll create a logical vector indicating which values of `happy` are *invalid* using the `%in%` operation. Because we want to see which values are *invalid*, we'll add the `== FALSE` condition (If we don't, the index will tell us which values *are* valid).

```
# Which values of happy are NOT in the set 1:5?
invalid <- (happy %in% 1:5) == FALSE
invalid
##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

Now that we have a logical index `invalid` telling us which values are invalid (that is, not in the set 1 through 5), we'll index `happy` with `invalid`, and assign the invalid values as NA:

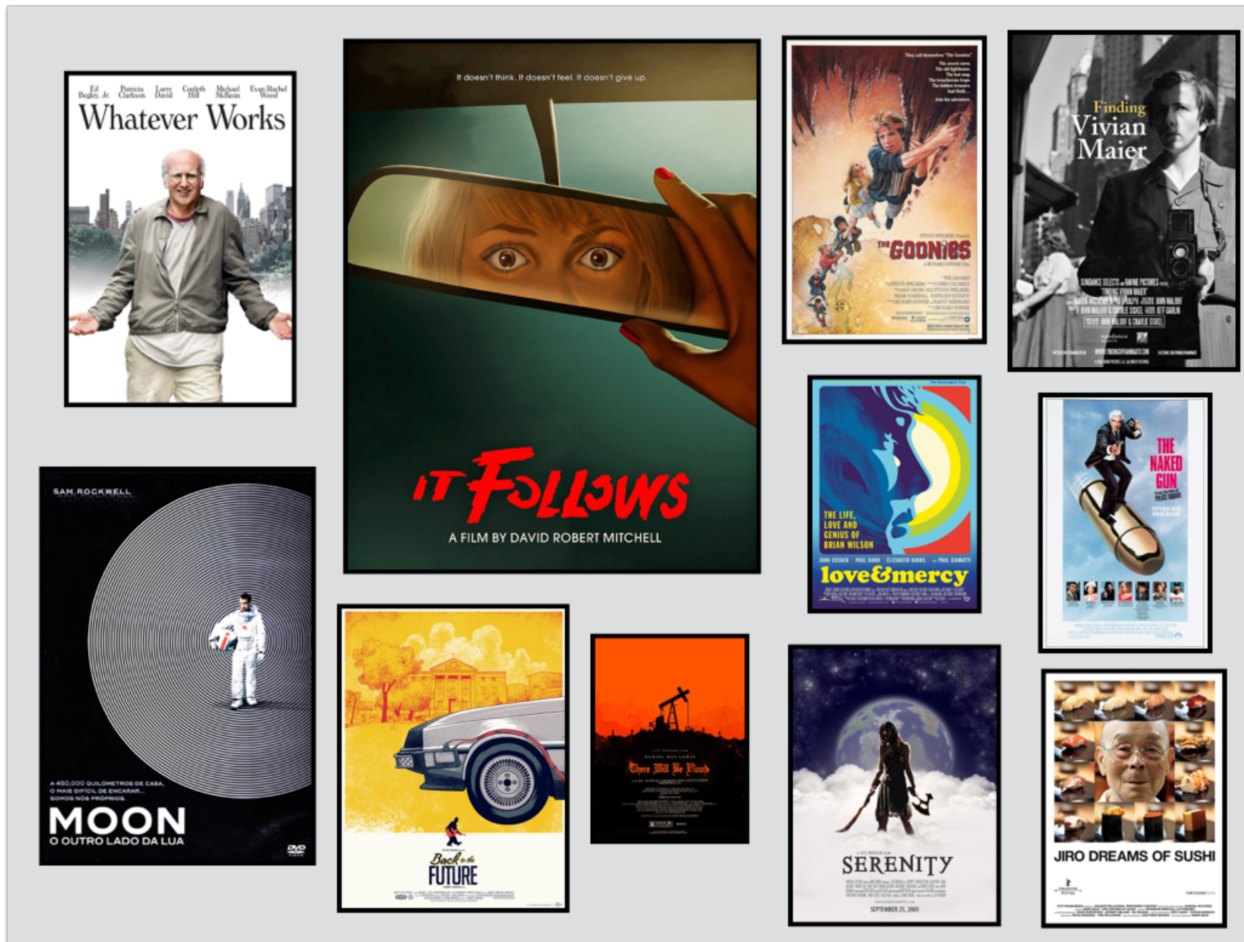
```
# Convert any invalid values in happy to NA
happy[invalid] <- NA
happy
##  [1]  1  4  2 NA  2  3 NA  3  2 NA
```

We can also recode all the invalid values of `happy` in one line as follows:

```
# Convert all values of happy that are NOT integers from 1 to 5 to NA
happy[(happy %in% 1:5) == FALSE] <- NA
```

As you can see, `happy` now has NAs for previously invalid values. Now we can take a `mean()` of the vector and see the mean of the valid responses.

```
# Include na.rm = TRUE to ignore NA values
mean(happy, na.rm = TRUE)
## [1] 2.4
```



7.4 Test your R Might!: Movie data

Table 7.2 contains data about 10 of my favorite movies.

0. Create new data vectors for each column.
1. What is the name of the 10th movie in the list?
2. What are the genres of the first 4 movies?
3. Some joker put Spice World in the movie names – it should be "The Naked Gun" Please correct the name.
4. What were the names of the movies made before 1990?
5. How many movies were Dramas? What percent of the 10 movies were Dramas?
6. One of the values in the `time` vector is invalid. Convert any invalid values in this vector to NA. Then, calculate the mean movie time
7. What were the names of the Comedy movies? What were their boxoffice totals? (Two separate questions)
8. What were the names of the movies that made less than $50 Million dollars AND were Comedies?
9. What was the median boxoffice revenue of movies rated either G or PG?
10. What percent of the movies were rated R OR were comedies?

Table 7.2: Some of my favorite movies

movie	year	boxoffice	genre	time	rating
Whatever Works	2009	35.0	Comedy	92	PG-13
It Follows	2015	15.0	Horror	97	R
Love and Mercy	2015	15.0	Drama	120	R
The Goonies	1985	62.0	Adventure	90	PG
Jiro Dreams of Sushi	2012	3.0	Documentary	81	G
There Will be Blood	2007	10.0	Drama	158	R
Moon	2009	321.0	Science Fiction	97	R
Spice World	1988	79.0	Comedy	-84	PG-13
Serenity	2005	39.0	Science Fiction	119	PG-13
Finding Vivian Maier	2014	1.5	Documentary	84	Unrated

Chapter 8

Matrices and Dataframes

```r
# -----------------------------
# Basic dataframe operations
# -----------------------------

# Create a dataframe of boat sale data called bsale
bsale <- data.frame(name = c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j"),
                    color = c("black", "green", "pink", "blue", "blue",
                              "green", "green", "yellow", "black", "black"),
                    age = c(143, 53, 356, 23, 647, 24, 532, 43, 66, 86),
                    price = c(53, 87, 54, 66, 264, 32, 532, 58, 99, 132),
                    cost = c(52, 80, 20, 100, 189, 12, 520, 68, 80, 100),
                    stringsAsFactors = FALSE)   # Don't convert strings to factors!

# Explore the bsale dataset:
head(bsale)     # Show me the first few rows
str(bsale)      # Show me the structure of the data
View(bsale)     # Open the data in a new window
names(bsale)    # What are the names of the columns?
nrow(bsale)     # How many rows are there in the data?

# Calculating statistics from column vectors
mean(bsale$age)       # What was the mean age?
table(bsale$color)    # How many boats were there of each color?
max(bsale$price)      # What was the maximum price?

# Adding new columns
bsale$id <- 1:nrow(bsale)
bsale$age.decades <- bsale$age / 10
bsale$profit <- bsale$price - bsale$cost

# What was the mean price of green boats?
with(bsale, mean(price[color == "green"]))

# What were the names of boats older than 100 years?
with(bsale, name[age > 100])

# What percent of black boats had a positive profit?
```

Figure 8.1: Did you actually think I could talk about matrices without a Matrix reference?!

```
with(subset(bsale, color == "black"), mean(profit > 0))

# Save only the price and cost columns in a new dataframe
bsale.2 <- bsale[c("price", "cost")]

# Change the names of the columns to "p" and "c"
names(bsale.2) <- c("p", "c")

# Create a dataframe called old.black.bsale containing only data from black boats older than 50 years
old.black.bsale <- subset(bsale, color == "black" & age > 50)
```

8.1 What are matrices and dataframes?

By now, you should be comfortable with scalar and vector objects. However, you may have noticed that neither object types are appropriate for storing lots of data – such as the results of a survey or experiment. Thankfully, R has two object types that represent large data structures much better: **matrices** and **dataframes**.

Matrices and dataframes are very similar to spreadsheets in Excel or data files in SPSS. Every matrix or dataframe contains rows (call that number m) and columns (n). Thus, wile a vector has 1 dimension (its length), matrices and dataframes both have 2-dimensions – representing their width and height. You can think of a matrix or dataframe as a combination of `n` vectors, where each vector has a length of `m`.

While matrices and dataframes look very similar, they aren't exactly the same. While a matrix can contain *either* character *or* numeric columns, a dataframe can contain *both* numeric and character columns. Because dataframes are more flexible, most real-world datasets, such as surveys containing both numeric (e.g.; age, response times) and character (e.g.; sex, favorite movie) data, will be stored as dataframes in R.

WTF – If dataframes are more flexible than matrices, why do we use matrices at all? The answer is that, because they are simpler, matrices take up less computational space than dataframes. Additionally, some functions require matrices as inputs to ensure that they work correctly.

In the next section, we'll cover the most common functions for creating matrix and dataframe objects. We'll then move on to functions that take matrices and dataframes as inputs.

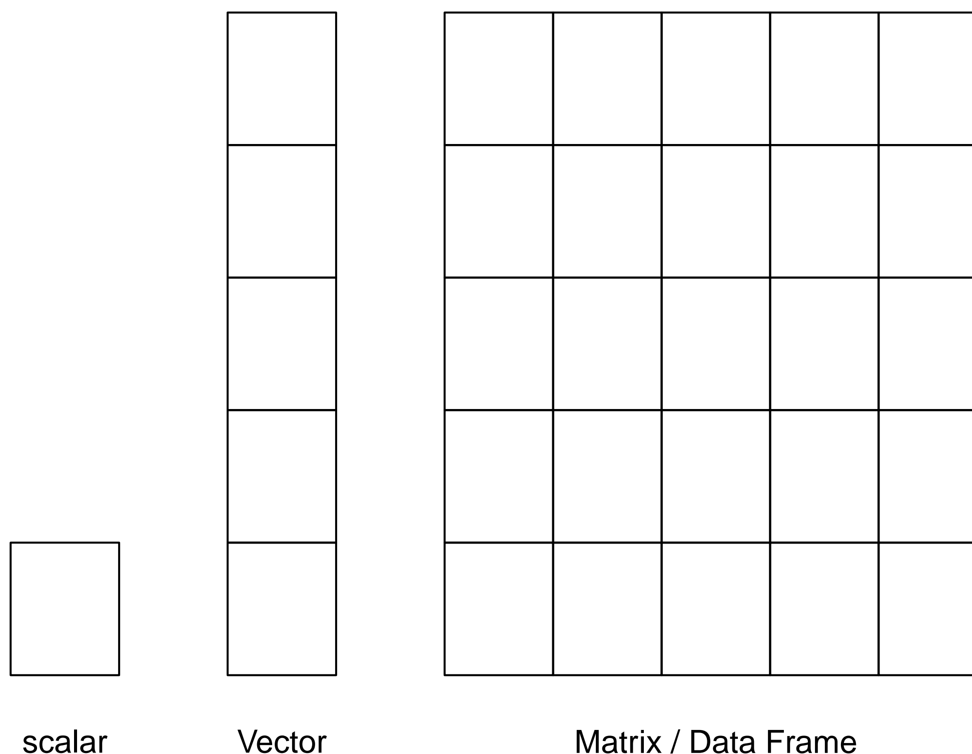


Figure 8.2: scalar, Vector, MATRIX

8.2 Creating matrices and dataframes

There are a number of ways to create your own matrix and dataframe objects in R. The most common functions are presented in Table 8.1. Because matrices and dataframes are just combinations of vectors, each function takes one or more vectors as inputs, and returns a matrix or a dataframe.

Table 8.1: Functions to create matrices and dataframes.

Function	Description	Example
`cbind(a, b, c)`	Combine vectors as columns in a matrix	`cbind(1:5, 6:10, 11:15)`
`rbind(a, b, c)`	Combine vectors as rows in a matrix	`rbind(1:5, 6:10, 11:15)`
`matrix(x, nrow, ncol, byrow)`	Create a matrix from a vector `x`	`matrix(x = 1:12, nrow = 3, ncol = 4)`
`data.frame()`	Create a dataframe from named columns	`data.frame("age" = c(19, 21), sex = c("m", "f"))`

8.2.1 `cbind()`, `rbind()`

`cbind()` and `rbind()` both create matrices by combining several vectors of the same length. `cbind()` combines vectors as columns, while `rbind()` combines them as rows.

Let's use these functions to create a matrix with the numbers 1 through 30. First, we'll create three vectors of length 5, then we'll combine them into one matrix. As you will see, the `cbind()` function will combine the vectors as columns in the final matrix, while the `rbind()` function will combine them as rows.

```
x <- 1:5
y <- 6:10
z <- 11:15

# Create a matrix where x, y and z are columns
cbind(x, y, z)
```

```
# Creating a matrix with numeric and character columns will make everything a character:

cbind(c(1, 2, 3, 4, 5),
      c("a", "b", "c", "d", "e"))
##      [,1] [,2]
## [1,] "1"  "a"
## [2,] "2"  "b"
## [3,] "3"  "c"
## [4,] "4"  "d"
## [5,] "5"  "e"
```

The `matrix()` function creates a matrix form a single vector of data. The function has 4 main inputs: `data` – a vector of data, `nrow` – the number of rows you want in the matrix, and `ncol` – the number of columns you want in the matrix, and `byrow` – a logical value indicating whether you want to fill the matrix by rows. Check out the help menu for the matrix function (‘?matrix) to see some additional inputs.

Let's use the `matrix()` function to re-create a matrix containing the values from 1 to 10.

```
# Create a matrix of the integers 1:10,
#  with 5 rows and 2 columns

matrix(data = 1:10,
       nrow = 5,
       ncol = 2)
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10

# Now with 2 rows and 5 columns
matrix(data = 1:10,
       nrow = 2,
       ncol = 5)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10

# Now with 2 rows and 5 columns, but fill by row instead of columns
matrix(data = 1:10,
       nrow = 2,
       ncol = 5,
       byrow = TRUE)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
```

8.2.3 data.frame()

To create a dataframe from vectors, use the `data.frame()` function. The `data.frame()` function works very similarly to `cbind()` – the only difference is that in `data.frame()` you specify names to each of the columns as you define them. Again, unlike matrices, dataframes can contain *both* string vectors and

numeric vectors within the same object. Because they are more flexible than matrices, most large datasets in R will be stored as dataframes.

Let's create a simple dataframe called **survey** using the **data.frame()** function with a mixture of text and numeric columns:

```
# Create a dataframe of survey data

survey <- data.frame("index" = c(1, 2, 3, 4, 5),
                     "sex" = c("m", "m", "m", "f", "f"),
                     "age" = c(99, 46, 23, 54, 23))
survey
##   index sex age
## 1     1   m  99
## 2     2   m  46
## 3     3   m  23
## 4     4   f  54
## 5     5   f  23
```

8.2.3.1 stringsAsFactors = FALSE

There is one key argument to `data.frame()` and similar functions called `stringsAsFactors`. By default, the `data.frame()` function will automatically convert any string columns to a specific type of object called a **factor** in R. A factor is a nominal variable that has a well-specified possible set of values that it can take on. For example, one can create a factor `sex` that can *only* take on the values `"male"` and `"female"`.

However, as I'm sure you'll discover, having R automatically convert your string data to factors can lead to lots of strange results. For example: if you have a factor of sex data, but then you want to add a new value called `other`, R will yell at you and return an error. I *hate*, *hate*, *HATE* when this happens. While there are very, very rare cases when I find factors useful, I almost always don't want or need them. For this reason, I avoid them at all costs.

To tell R to *not* convert your string columns to factors, you need to include the argument `stringsAsFactors = FALSE` when using functions such as `data.frame()`

For example, let's look at the classes of the columns in the dataframe `survey` that we just created using the `str()` function (we'll go over this function in section XXX)

```
# Show me the structure of the survey dataframe
str(survey)
## 'data.frame':    5 obs. of  3 variables:
##  $ index: num  1 2 3 4 5
##  $ sex  : Factor w/ 2 levels "f","m": 2 2 2 1 1
##  $ age  : num  99 46 23 54 23
```

AAAAA!!! R has converted the column `sex` to a factor with *only* two possible levels! This can cause major problems later! Let's create the dataframe again using the argument `stringsAsFactors = FALSE` to make sure that this doesn't happen:

```
# Create a dataframe of survey data WITHOUT factors
survey <- data.frame("index" = c(1, 2, 3, 4, 5),
                     "sex" = c("m", "m", "m", "f", "f"),
                     "age" = c(99, 46, 23, 54, 23),
                     stringsAsFactors = FALSE)
```

Now let's look at the new version and make sure there are no factors:

```
# Print the result (it looks the same as before)
survey
##   index sex age
## 1     1   m  99
## 2     2   m  46
## 3     3   m  23
## 4     4   f  54
## 5     5   f  23

# Look at the structure: no more factors!
str(survey)
## 'data.frame':    5 obs. of  3 variables:
##  $ index: num  1 2 3 4 5
##  $ sex  : chr  "m" "m" "m" "f" ...
##  $ age  : num  99 46 23 54 23
```

8.2.4 Dataframes pre-loaded in R

Now you know how to use functions like `cbind()` and `data.frame()` to manually create your own matrices and dataframes in R. However, for demonstration purposes, it's frequently easier to use existing dataframes rather than always having to create your own. Thankfully, R has us covered: R has several datasets that come pre-installed in a package called `datasets` – you don't need to install this package, it's included in the base R software. While you probably won't make any major scientific discoveries with these datasets, they allow all R users to test and compare code on the same sets of data. To see a complete list of all the datasets included in the `datasets` package, run the code: `library(help = "datasets")`. Table 8.2 shows a few datasets that we will be using in future examples:

Table 8.2: A few datasets you can access in R.

Dataset	Description	Rows	Columns
`ChickWeight`	Experiment on the effect of diet on early growth of chicks.	578	4
`InsectSprays`	The counts of insects in agricultural experimental units treated with different insecticides.	72	2
`ToothGrowth`	Length of odontoblasts (cells responsible for tooth growth) in 60 guinea pigs.	60	3
`PlantGrowth`	Results from an experiment to compare yields (as measured by dried weight of plants) obtained under a control and two different treatment conditions.	30	2

8.3 Matrix and dataframe functions

R has lots of functions for viewing matrices and dataframes and returning information about them. Table 8.3 shows some of the most common:

Table 8.3: Important functions for understanding matrices and dataframes.

Function	Description
`head(x), tail(x)`	Print the first few rows (or last few rows).
`View(x)`	Open the entire object in a new window
`nrow(x), ncol(x), dim(x)`	Count the number of rows and columns
`rownames(), colnames(), names()`	Show the row (or column) names
`str(x), summary(x)`	Show the structure of the dataframe (ie., dimensions and classes) and summary statistics

8.3.1 head(), tail(), View()

To see the first few rows of a dataframe, use `head()`, to see the last few rows, use `tail()`

```
# head() shows the first few rows
head(ChickWeight)
## Grouped Data: weight ~ Time | Chick
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
## 3     59    4     1    1
## 4     64    6     1    1
## 5     76    8     1    1
## 6     93   10     1    1

# tail() shows he last few rows
tail(ChickWeight)
## Grouped Data: weight ~ Time | Chick
##     weight Time Chick Diet
## 573    155   12    50    4
## 574    175   14    50    4
## 575    205   16    50    4
## 576    234   18    50    4
## 577    264   20    50    4
## 578    264   21    50    4
```

To see an entire dataframe in a separate window that looks like spreadsheet, use `View()`

```
# View() opens the entire dataframe in a new window
View(ChickWeight)
```

When you run `View()`, you'll see a new window like the one in Figure 8.3

8.3.2 summary(), str()

To get summary statistics on all columns in a dataframe, use the `summary()` function:

```
# Print summary statistics of ToothGrowth to the console
summary(ToothGrowth)
##       len      supp         dose          len.cm        index
##  Min.   : 4   OJ:30   Min.   :0.50   Min.   :0.4   Min.   : 1
```

	weight	Time	Chick	Diet
1	42	0	1	1
2	51	2	1	1
3	59	4	1	1
4	64	6	1	1
5	76	8	1	1
6	93	10	1	1
7	106	12	1	1
8	125	14	1	1
9	149	16	1	1
10	171	18	1	1
11	199	20	1	1
12	205	21	1	1
13	40	0	2	1

Showing 1 to 14 of 578 entries

Figure 8.3: Screenshot of the window from View(ChickWeight). You can use this window to visually sort and filter the data to get an idea of how it looks, but you can't add or remove data and nothing you do will actually change the dataframe.

```
##  1st Qu.:13   VC:30   1st Qu.:0.50   1st Qu.:1.3   1st Qu.:16
##  Median :19           Median :1.00   Median :1.9   Median :30
##  Mean   :19           Mean   :1.17   Mean   :1.9   Mean   :30
##  3rd Qu.:25           3rd Qu.:2.00   3rd Qu.:2.5   3rd Qu.:45
##  Max.   :34           Max.   :2.00   Max.   :3.4   Max.   :60
```

To learn about the classes of columns in a dataframe, in addition to some other summary information, use the `str()` (structure) function. This function returns information for more advanced R users, so don't worry if the output looks confusing.

```
# Print additional information about ToothGrowth to the console
str(ToothGrowth)
## 'data.frame':    60 obs. of  5 variables:
##  $ len   : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
##  $ supp  : Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
##  $ dose  : num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
##  $ len.cm: num  0.42 1.15 0.73 0.58 0.64 1 1.12 1.12 0.52 0.7 ...
##  $ index : int  1 2 3 4 5 6 7 8 9 10 ...
```

Here, we can see that `ToothGrowth` is a dataframe with 60 observations (ie., rows) and 5 variables (ie., columns). We can also see that the column names are `index`, `len`, `len.cm`, `supp`, and `dose`

8.4 Dataframe column names

One of the nice things about dataframes is that each column will have a name. You can use these name to access specific columns by name without having to know which column number it is.

To access the names of a dataframe, use the function `names()`. This will return a string vector with the names of the dataframe. Let's use `names()` to get the names of the `ToothGrowth` dataframe:

```
# What are the names of columns in the ToothGrowth dataframe?
names(ToothGrowth)
## [1] "len"    "supp"   "dose"   "len.cm" "index"
```

To access a specific column in a dataframe by name, you use the `$` operator in the form `df$name` where `df` is the name of the dataframe, and `name` is the name of the column you are interested in. This operation will then return the column you want as a vector.

Let's use the `$` operator to get a vector of just the length column (called `len`) from the `ToothGrowth` dataframe:

```
# Return the len column of ToothGrowth
ToothGrowth$len
##  [1]  4.2 11.5  7.3  5.8  6.4 10.0 11.2 11.2  5.2  7.0 16.5 16.5 15.2 17.3
## [15] 22.5 17.3 13.6 14.5 18.8 15.5 23.6 18.5 33.9 25.5 26.4 32.5 26.7 21.5
## [29] 23.3 29.5 15.2 21.5 17.6  9.7 14.5 10.0  8.2  9.4 16.5  9.7 19.7 23.3
## [43] 23.6 26.4 20.0 25.2 25.8 21.2 14.5 27.3 25.5 26.4 22.4 24.5 24.8 30.9
## [57] 26.4 27.3 29.4 23.0
```

Because the `$` operator returns a vector, you can easily calculate descriptive statistics on columns of a dataframe by applying your favorite vector function (like `mean()` or `table()`) to a column using `$`. Let's calculate the mean tooth length with `mean()`, and the frequency of each supplement with `table()`:

```
# What is the mean of the len column of ToothGrowth?
mean(ToothGrowth$len)
## [1] 19

# Give me a table of the supp column of ToothGrowth.
table(ToothGrowth$supp)
##
## OJ VC
## 30 30
```

If you want to access several columns by name, you can forgo the $ operator, and put a character vector of column names in brackets:

```
# Give me the len AND supp columns of ToothGrowth
head(ToothGrowth[c("len", "supp")])
##    len supp
## 1  4.2   VC
## 2 11.5   VC
## 3  7.3   VC
## 4  5.8   VC
## 5  6.4   VC
## 6 10.0   VC
```

8.4.1 Adding new columns

You can add new columns to a dataframe using the `$` and assignment `<-` operators. To do this, just use the `df$name` notation and assign a new vector of data to it.

For example, let's create a dataframe called `survey` with two columns: `index` and `age`:

```
# Create a new dataframe called survey
survey <- data.frame("index" = c(1, 2, 3, 4, 5),
                     "age" = c(24, 25, 42, 56, 22))

survey
##   index age
## 1     1  24
## 2     2  25
```

```
## 3     3  42
## 4     4  56
## 5     5  22
```

Now, let's add a new column called `sex` with a vector of sex data:

```
# Add a new column called sex to survey
survey$sex <- c("m", "m", "f", "f", "m")
```

Here's the result

```
# survey with new sex column
survey
##   index age sex
## 1     1  24   m
## 2     2  25   m
## 3     3  42   f
## 4     4  56   f
## 5     5  22   m
```

As you can see, `survey` has a new column with the name `sex` with the values we specified earlier.

8.4.2 Changing column names

To change the name of a column in a dataframe, just use a combination of the `names()` function, indexing, and reassignment.

```
# Change name of 1st column of df to "a"
names(df)[1] <- "a"

# Change name of 2nd column of df to "b"
names(df)[2] <- "b"
```

For example, let's change the name of the first column of `survey` from `index` to `participant.number`

```
# Change the name of the first column of survey to "participant.number"
names(survey)[1] <- "participant.number"
survey
##   participant.number age sex
## 1                  1  24   m
## 2                  2  25   m
## 3                  3  42   f
## 4                  4  56   f
## 5                  5  22   m
```

Warning!!!: Change column names with logical indexing to avoid errors!

Now, there is one major potential problem with my method above – I had to manually enter the value of 1. But what if the column I want to change isn't in the first column (either because I typed it wrong or because the order of the columns changed)? This could lead to serious problems later on.

To avoid these issues, it's better to change column names using a logical vector using the format `names(df)[names(df) == "old.name"] <- "new.name"`. Here's how to read this: "Change the names of `df`, but only where the original name was `"old.name"`, to `"new.name"`.

Let's use logical indexing to change the name of the column `survey$age` to `survey$years`:

```
# Change the column name from age to age.years
names(survey)[names(survey) == "age"] <- "years"
```

Figure 8.4: Slicing and dicing data. The turnip represents your data, and the knife represents indexing with brackets, or subsetting functions like subset(). The red-eyed clown holding the knife is just off camera.

```
survey
##   participant.number years sex
## 1                  1    24   m
## 2                  2    25   m
## 3                  3    42   f
## 4                  4    56   f
## 5                  5    22   m
```

8.5 Slicing dataframes

Once you have a dataset stored as a matrix or dataframe in R, you'll want to start accessing specific parts of the data based on some criteria. For example, if your dataset contains the result of an experiment comparing different experimental groups, you'll want to calculate statistics for each experimental group separately. The process of selecting specific rows and columns of data based on some criteria is commonly known as *slicing*.

8.5.1 Slicing with [,]

Just like vectors, you can access specific data in dataframes using brackets. But now, instead of just using one indexing vector, we use two indexing vectors: one for the rows and one for the columns. To do this, use the notation `data[rows, columns]`, where `rows` and `columns` are vectors of integers.

```
# Return row 1
df[1, ]

# Return column 5
df[, 5]

# Rows 1:5 and column 2
df[1:5, 2]
```

Figure 8.5: Ah the ToothGrowth dataframe. Yes, one of the dataframes stored in R contains data from an experiment testing the effectiveness of different doses of Vitamin C supplements on the growth of guinea pig teeth. The images I found by Googling "guinea pig teeth" were all pretty horrifying, so let's just go with this one.

Table 8.4: First few rows of the ToothGrowth dataframe.

len	supp	dose	len.cm	index
4.2	VC	0.5	0.42	1
11.5	VC	0.5	1.15	2
7.3	VC	0.5	0.73	3
5.8	VC	0.5	0.58	4
6.4	VC	0.5	0.64	5
10.0	VC	0.5	1.00	6

Let's try indexing the `ToothGrowth` dataframe. Again, the `ToothGrowth` dataframe represents the results of a study testing the effectiveness of different types of supplements on the length of guinea pig's teeth. First, let's look at the entries in rows 1 through 5, and column 1:

```
# Give me the rows 1-6 and column 1 of ToothGrowth
ToothGrowth[1:6, 1]
## [1]  4.2 11.5  7.3  5.8  6.4 10.0
```

Because the first column is `len`, the primary dependent measure, this means that the tooth lengths in the first 6 observations are 4.2, 11.5, 7.3, 5.8, 6.4, 10.

Of course, you can index matrices and dataframes with longer vectors to get more data. Now, let's look at the first 3 rows of columns 1 and 3:

```
# Give me rows 1-3 and columns 1 and 3 of ToothGrowth
ToothGrowth[1:3, c(1,3)]
##    len dose
## 1  4.2  0.5
## 2 11.5  0.5
## 3  7.3  0.5
```

If you want to look at an entire row or an entire column of a matrix or dataframe, you can leave the corresponding index blank. For example, to see the entire 1st row of the `ToothGrowth` dataframe, we can set the row index to 1, and leave the column index blank:

```
# Give me the 1st row (and all columns) of ToothGrowth
ToothGrowth[1, ]
##   len supp dose len.cm index
## 1 4.2   VC  0.5   0.42     1
```

Similarly, to get the entire 2nd column, set the column index to 2 and leave the row index blank:

```
# Give me the 2nd column (and all rows) of ToothGrowth
ToothGrowth[, 2]
```

Figure 8.6: The subset() function is like a lightsaber. An elegant function from a more civilized age.

```
##  [1] VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC VC
## [24] VC VC VC VC VC VC VC OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ
## [47] OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ OJ
## Levels: OJ VC
```

Many, if not all, of the analyses you will be doing will be on subsets of data, rather than entire datasets. For example, if you have data from an experiment, you may wish to calculate the mean of participants in one group separately from another. To do this, we'll use *subsetting* – selecting subsets of data based on some criteria. To do this, we can use one of two methods: indexing with logical vectors, or the `subset()` function. We'll start with logical indexing first.

8.5.2 Slicing with logical vectors

Indexing dataframes with logical vectors is almost identical to indexing single vectors. First, we create a logical vector containing only TRUE and FALSE values. Next, we index a dataframe (typically the rows) using the logical vector to return *only* values for which the logical vector is TRUE.

For example, to create a new dataframe called `ToothGrowth.VC` containing only data from the guinea pigs who were given the VC supplement, we'd run the following code:

```
# Create a new df with only the rows of ToothGrowth
#  where supp equals VC
ToothGrowth.VC <- ToothGrowth[ToothGrowth$supp == "VC", ]
```

Of course, just like we did with vectors, we can make logical vectors based on multiple criteria – and then index a dataframe based on those criteria. For example, let's create a dataframe called `ToothGrowth.OJ.a` that contains data from the guinea pigs who were given an OJ supplement with a dose less than 1.0:

```
# Create a new df with only the rows of ToothGrowth
#  where supp equals OJ and dose < 1

ToothGrowth.OJ.a <- ToothGrowth[ToothGrowth$supp == "OJ" &
                                ToothGrowth$dose < 1, ]
```

Indexing with brackets is the standard way to slice and dice dataframes. However, the code can get a bit messy. A more elegant method is to use the `subset()` function.

8.5.3 Slicing with subset()

The `subset()` function is one of the most useful data management functions in R. It allows you to slice and dice datasets just like you would with brackets, but the code is much easier to write: Table 8.5 shows the main arguments to the `subset()` function:

Table 8.5: Main arguments for the `subset()` function.

Argument	Description
x	A dataframe you want to subset
subset	A logical vector indicating the rows to keep
select	The columns you want to keep

Let's use the `subset()` function to create a new, subsetted dataset from the `ToothGrowth` dataframe containing data from guinea pigs who had a tooth length less than 20cm (`len < 20`), given the OJ supplement (`supp == "OJ"`), and with a dose greater than or equal to 1 (`dose >= 1`):

```
# Get rows of ToothGrowth where len < 20 AND supp == "OJ" AND dose >= 1
subset(x = ToothGrowth,
      subset = len < 20 &
               supp == "OJ" &
               dose >= 1)
##    len supp dose len.cm index
## 41  20   OJ    1    2.0    41
## 49  14   OJ    1    1.4    49
```

As you can see, there were only two cases that satisfied all 3 of our selection criteria.

In the example above, I didn't specify an input to the `select` argument because I wanted all columns. However, if you just want certain columns, you can just name the columns you want in the `select` argument:

```
# Get rows of ToothGrowth where len > 30 AND supp == "VC", but only return the len and dose columns
subset(x = ToothGrowth,
    subset = len > 30 & supp == "VC",
    select = c(len, dose))
##    len dose
## 23  34    2
## 26  32    2
```

8.6 Combining slicing with functions

Now that you know how to slice and dice dataframes using indexing and `subset()`, you can easily combine slicing and dicing with statistical functions to calculate summary statistics on groups of data. For example, the following code will calculate the mean tooth length of guinea pigs with the OJ supplement using the `subset()` function:

```
# What is the mean tooth length of Guinea pigs given OJ?

# Step 1: Create a subsettted dataframe called oj

oj <- subset(x = ToothGrowth,
             subset = supp == "OJ")
```

```
# Step 2: Calculate the mean of the len column from
#  the new subsetted dataset

mean(oj$len)
## [1] 21
```

We can also get the same solution using logical indexing:

```
# Step 1: Create a subsettted dataframe called oj
oj <- ToothGrowth[ToothGrowth$supp == "OJ",]

# Step 2: Calculate the mean of the len column from
#  the new subsetted dataset
mean(oj$len)
## [1] 21
```

Or heck, we can do it all in one line by only referring to column vectors:

```
mean(ToothGrowth$len[ToothGrowth$supp == "OJ"])
## [1] 21
```

As you can see, R allows for many methods to accomplish the same task. The choice is up to you.

8.6.1 with()

The `with()` function helps to save you some typing when you are using multiple columns from a dataframe. Specifically, it allows you to specify a dataframe (or any other object in R) once at the beginning of a line – then, for every object you refer to in the code in that line, R will assume you're referring to that object in an expression.

For example, let's create a dataframe called `health` with some health information:

```
health <- data.frame("age" = c(32, 24, 43, 19, 43),
                     "height" = c(1.75, 1.65, 1.50, 1.92, 1.80),
                     "weight" = c(70, 65, 62, 79, 85))

health
##   age height weight
## 1  32    1.8     70
## 2  24    1.6     65
## 3  43    1.5     62
## 4  19    1.9     79
## 5  43    1.8     85
```

Now let's say we want to add a new column called `bmi` which represents a person's body mass index. The formula for bmi is $bmi = \frac{height}{weight^2} \times 703$. If we wanted to calculate the bmi of each person, we'd need to write `health$height / health$weight ^ 2`:

```
# Calculate bmi
health$height / health$weight ^ 2
## [1] 0.00036 0.00039 0.00039 0.00031 0.00025
```

As you can see, we have to retype the name of the dataframe for each column. However, using the `with()` function, we can make it a bit easier by saying the name of the dataframe once.

Figure 8.7: This is a lesser-known superhero named Maggott who could 'transform his body to get superhuman strength and endurance, but to do so he needed to release two huge parasitic worms from his stomach cavity and have them eat things' (http://heavy.com/comedy/2010/04/the-20-worst-superheroes/). Yeah...I'm shocked this guy wasn't a hit.

```
# Save typing by using with()
with(health, height / weight ^ 2)
## [1] 0.00036 0.00039 0.00039 0.00031 0.00025
```

As you can see, the results are identical. In this case, we didn't save so much typing. But if you are doing many calculations, then `with()` can save you a lot of typing. For example, contrast these two lines of code that perform identical calculations:

```
# Long code
health$weight + health$height / health$age + 2 * health$height
## [1] 74 68 65 83 89

# Short code that does the same thing
with(health, weight + height / age + 2 * height)
## [1] 74 68 65 83 89
```

8.7 Test your R might! Pirates and superheroes

The following table shows the results of a survey of 10 pirates. In addition to some basic demographic information, the survey asked each pirate "What is your favorite superhero?"" and "How many tattoos do you have?""

Name	Sex	Age	Superhero	Tattoos
Astrid	F	30	Batman	11
Lea	F	25	Superman	15
Sarina	F	25	Batman	12
Remon	M	29	Spiderman	5
Letizia	F	22	Batman	65
Babice	F	22	Antman	3
Jonas	M	35	Batman	9
Wendy	F	19	Superman	13
Niveditha	F	32	Maggott	900
Gioia	F	21	Superman	0

1. Combine the data into a single dataframe. Complete all the following exercises from the dataframe!
2. What is the median age of the 10 pirates?
3. What was the mean age of female and male pirates separately?
4. What was the most number of tattoos owned by a male pirate?
5. What percent of pirates under the age of 32 were female?
6. What percent of female pirates are under the age of 32?
7. Add a new column to the dataframe called `tattoos.per.year` which shows how many tattoos each pirate has for each year in their life.
8. Which pirate had the most number of tattoos per year?
9. What are the names of the female pirates whose favorite superhero is Superman?
10. What was the median number of tattoos of pirates over the age of 20 whose favorite superhero is Spiderman?

Chapter 9

Importing, saving and managing data

Remember way back in Chapter 2 (you know...back when we first met...we were so young and full of excitement then...sorry, now I'm getting emotional...let's move on...) when I said everything in R is an object? Well, that's still true. In this chapter, we'll cover the basics of R object management. We'll cover how to load new objects like external datasets into R, how to manage the objects that you already have, and how to export objects from R into external files that you can share with other people or store for your own future use.

9.1 Workspace management functions

Here are some functions helpful for managing your workspace that we'll go over in this chapter:

Table 9.1: Functions for managing your workspace, working directory, and writing data from R as `.txt` or `.RData` files, and reading files into R

Code	Description
`ls()`	Display all objects in the current workspace
`rm(a, b, ..)`	Removes the objects `a`, `b`... from your workspace
`rm(list = ls())`	Removes *all* objects in your workspace
`getwd()`	Returns the current working directory
`setwd(file = "dir)`	Changes the working directory to a specified file location
`list.files()`	Returns the names of all files in the working directory
`write.table(x, file = "mydata.txt", sep)`	writes the object `x` to a text file called `mydata.txt`. Define how the columns will be separated with `sep` (e.g.; `sep = ","` for a comma–separated file, and `sep = \t"` for a tab–separated file).
`save(a, b, .., file = "myimage.RData)`	Saves objects `a`, `b`, ... to `myimage.RData`
`save.image(file = "myimage.RData")`	Saves *all* objects in your workspace to `myimage.RData`
`load(file = "myimage.RData")`	Loads objects in the file `myimage.RData`

Code	Description
`read.table(file = "mydata.txt", sep, header)`	Reads a text file called `mydata.txt`, define how columns are separated with `sep` (e.g. `sep = ","` for comma-delimited files, and `sep = "\t"` for tab-delimited files), and whether there is a header column with `header = TRUE`

9.1.1 Why object and file management is so important

Your computer is a maze of folders, files, and selfies (see Figure 9.2). Outside of R, when you want to open a specific file, you probably open up an explorer window that allows you to visually search through the folders on your computer. Or, maybe you select recent files, or type the name of the file in a search box to let your computer do the searching for you. While this system usually works for non-programming tasks, it is a no-go for R. Why? Well, the main problem is that all of these methods require you to *visually* scan your folders and move your mouse to select folders and files that match what you are looking for. When you are programming in R, you need to specify *all* steps in your analyses in a way that can be easily replicated by others and your future self. This means you can't just say: "Find this one file I emailed to myself a week ago" or "Look for a file that looks something like `experimentAversion3.txt`." Instead, need to be able to write R code that tells R *exactly* where to find critical files – either on your computer or on the web.

To make this job easier, R uses *working directories*.

9.2 The working directory

The **working directory** is just a file path on your computer that sets the default location of any files you read into R, or save out of R. In other words, a working directory is like a little flag somewhere on your computer which is tied to a specific analysis project. If you ask R to import a dataset from a text file, or save a dataframe as a text file, it will assume that the file is inside of your working directory.

You can only have one working directory active at any given time. The active working directory is called your *current* working directory.

To see your current working directory, use `getwd()`:

```
# Print my current working directory
getwd()
## [1] "/Users/nphillips/Dropbox/manuscripts/YaRrr/YaRrr_bd"
```

As you can see, when I run this code, it tells me that my working directory is in a folder on my Desktop called `yarrr`. This means that when I try to read new files into R, or write files out of R, it will assume that I want to put them in this folder.

If you want to change your working directory, use the `setwd()` function. For example, if I wanted to change my working directory to an existing Dropbox folder called `yarrr`, I'd run the following code:

```
# Change my working directory to the following path
setwd(dir = "/Users/nphillips/Dropbox/yarrr")
```

9.3 Projects in RStudio

If you're using RStudio, you have the option of creating a new R **project**. A project is simply a working directory designated with a `.RProj` file. When you open a project (using File/Open Project in RStudio or

Figure 9.1: Your workspace – all the objects, functions, and delicious glue you've defined in your current session.

Figure 9.2: Your computer is probably so full of selfies like this that if you don't get organized, you may try to load this into your R session instead of your data file.

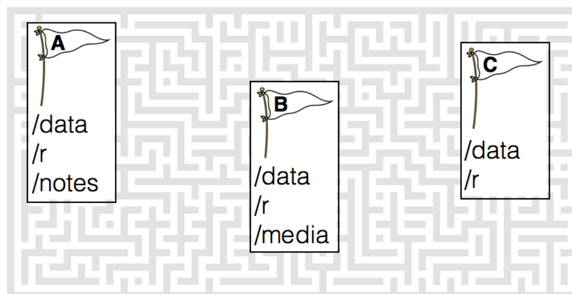


Figure 9.3: A working directory is like a flag on your computer that tells R where to start looking for your files related to a specific project. Each project should have its own folder with organized sub-folders.

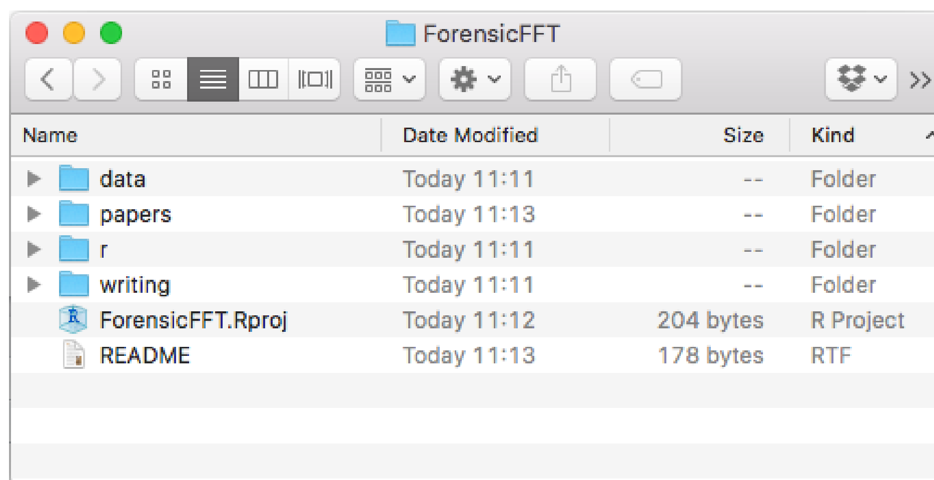


Figure 9.4: Here is the folder structure I use for the working directory in my R project called ForensicFFT. As you can see, it contains an .Rproj file generated by RStudio which sets this folder as the working directory. I also created a folder called r for R code, a folder called data for.txt and .RData files) among others.

by double–clicking on the .Rproj file outside of R), the working directory will automatically be set to the directory that the `.RProj` file is located in.

I recommend creating a new R Project whenever you are starting a new research project. Once you've created a new R project, you should immediately create folders in the directory which will contain your R code, data files, notes, and other material relevant to your project (you can do this outside of R on your computer, or in the Files window of RStudio). For example, you could create a folder called `R` that contains all of your R code, a folder called `data` that contains all your data (etc.). In Figure~9.4 you can see how my working directory looks for a project I am working on called ForensicFFT.

9.4 The workspace

The **workspace** (aka your **working environment**) represents all of the objects and functions you have either defined in the current session, or have loaded from a previous session. When you started RStudio for the first time, the working environment was empty because you hadn't created any new objects or functions. However, as you defined new objects and functions using the assignment operator `<-`, these new

objects were stored in your working environment. When you closed RStudio after defining new objects, you likely got a message asking you "Save workspace image…?"" This is RStudio's way of asking you if you want to save all the objects currently defined in your workspace as an **image file** on your computer.

9.4.1 ls()

If you want to see all the objects defined in your current workspace, use the `ls()` function.

```
# Print all the objects in my workspace
ls()
```

When I run `ls()` I received the following result:

```
## [1] "study1.df" "study2.df" "lm.study1" "lm.study2" "bf.study1"
```

The result above says that I have these 5 objects in my workspace. If I try to refer to an object not listed here, R will return an error. For example, if I try to print `study3.df` (which isn't in my workspace), I will receive the following error:

```
# Try to print study3.df
#  Error because study3.df is NOT in my current workspace
study3.df
```

Error: object 'study3.df' not found

If you receive this error, it's because the object you are referring to is not in your current workspace. 99% of the time, this happens because you mistyped the name of an object.

9.5 .RData files

The best way to store objects from R is with `.RData files`. `.RData` files are specific to R and can store as many objects as you'd like within a single file. Think about that. If you are conducting an analysis with 10 different dataframes and 5 hypothesis tests, you can save **all** of those objects in a single file called `ExperimentResults.RData`.

9.5.1 save()

To save selected objects into one `.RData` file, use the `save()` function. When you run the `save()` function with specific objects as arguments, (like `save(a, b, c, file = "myobjects.RData")` all of those objects will be saved in a single file called `myobjects.RData`

For example, let's create a few objects corresponding to a study.

```
# Create some objects that we'll save later
study1.df <- data.frame(id = 1:5,
                        sex = c("m", "m", "f", "f", "m"),
                        score = c(51, 20, 67, 52, 42))

score.by.sex <- aggregate(score ~ sex,
                          FUN = mean,
                          data = study1.df)

study1.htest <- t.test(score ~ sex,
                       data = study1.df)
```

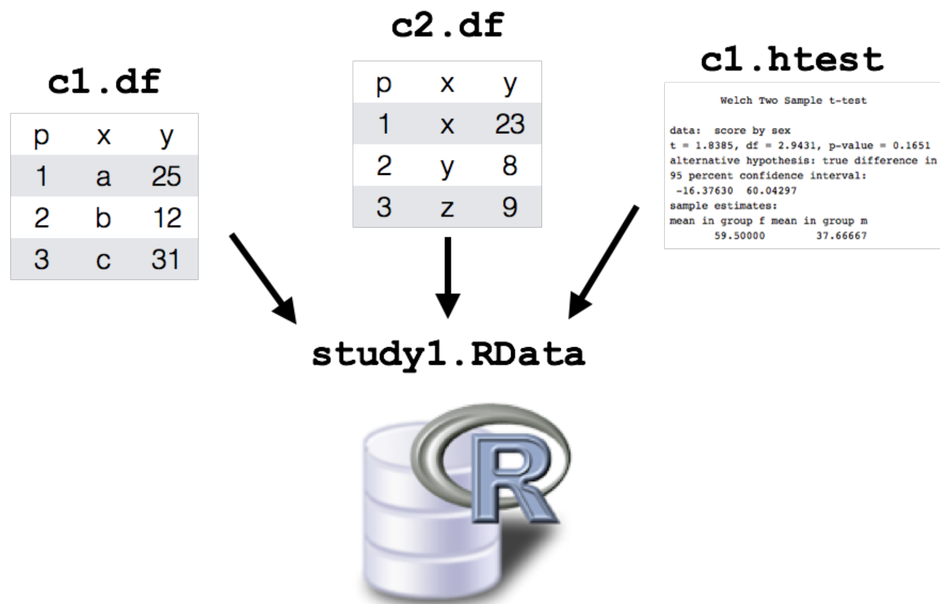


Figure 9.5: Saving multiple objects into a single .RData file.

Now that we've done all of this work, we want to save all three objects in an a file called `study1.RData` in the data folder of my current working directory. To do this, you can run the following

```
# Save two objects as a new .RData file
#   in the data folder of my current working directory
save(study1.df, score.by.sex, study1.htest,
     file = "data/study1.RData")
```

Once you do this, you should see the `study1.RData` file in the data folder of your working directory. This file now contains all of your objects that you can easily access later using the `load()` function (we'll go over this in a second...).

9.5.2 save.image()

If you have many objects that you want to save, then using `save` can be tedious as you'd have to type the name of every object. To save *all* the objects in your workspace as a .RData file, use the `save.image()` function. For example, to save my workspace in the `data` folder located in my working directory, I'd run the following:

```
# Save my workspace to complete_image.RData in th,e
#  data folder of my working directory
save.image(file = "data/projectimage.RData")
```

Now, the `projectimage.RData` file contains *all* objects in your current workspace.

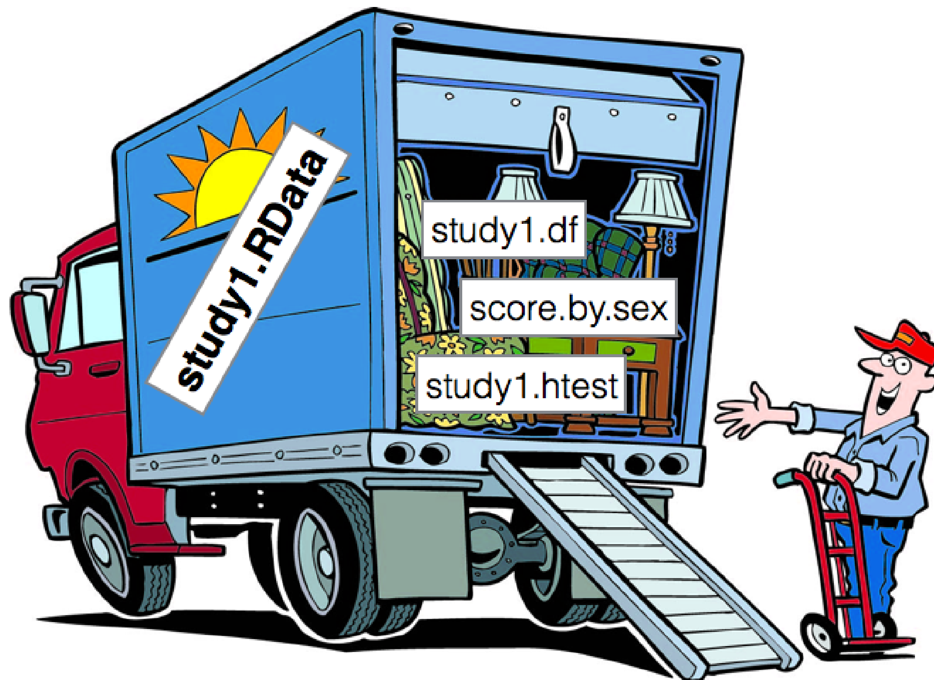


Figure 9.6: Our new study1.RData file is like a van filled with our objects.

9.5.3 load()

To load an `.RData` file, that is, to import all of the objects contained in the `.RData` file into your current workspace, use the `load()` function. For example, to load the three specific objects that I saved earlier (`study1.df`, `score.by.sex`, and `study1.htest`) in `study1.RData`, I'd run the following:

```
# Load objects in study1.RData into my workspace
load(file = "data/study1.RData")
```

To load all of the objects in the workspace that I just saved to the data folder in my working directory in `projectimage.RData`, I'd run the following:

```
# Load objects in projectimage.RData into my workspace
load(file = "data/projectimage.RData")
```

I hope you realize how awesome loading .RData files is. With R, you can store all of your objects, from dataframes to hypothesis tests, in a single `.RData` file. And then load them into any R session at any time using `load()`.

9.5.4 rm()

To remove objects from your workspace, use the `rm()` function. Why would you want to remove objects? At some points in your analyses, you may find that your workspace is filled up with one or more objects that you don't need – either because they're slowing down your computer, or because they're just distracting.

To remove specific objects, enter the objects as arguments to `rm()`. For example, to remove a huge dataframe called `huge.df`, I'd run the following;

```
# Remove huge.df from workspace
rm(huge.df)
```

If you want to remove *all* of the objects in your working directory, enter the argument `list = ls()`

```
# Remove ALL objects from workspace
rm(list = ls())
```

Important!!! Once you remove an object, you **cannot** get it back without running the code that originally generated the object! That is, you can't simply click 'Undo' to get an object back. Thankfully, if your R code is complete and well-documented, you should easily be able to either re-create a lost object (e.g.; the results of a regression analysis), or re-load it from an external file.

9.6 .txt files

While `.RData` files are great for saving R objects, sometimes you'll want to export data (usually dataframes) as a simple `.txt` text file that other programs, like Excel and **S**hitty **P**iece of **S**hitty **S**hit, can also read. To do this, use the `write.table()` function.

9.6.1 write.table()

Table 9.2: Arguments for the `write.table()` function that will save an object x (usually a data frame) as a .txt file.

Argument	Description
`x`	The object you want to write to a text file, usually a dataframe
`file`	The document's file path relative to the working directory unless specified otherwise. For example `file = "mydata.txt"` saves the text file directly in the working directory, while `file = "data/mydata.txt"` will save the data in an existing folder called `data` inside the working directory.You can also specify a full file path outside of your working directory (`file = "/Users/CaptainJack/Desktop/OctoberStudy/mydata.txt"`)
`sep`	A string indicating how the columns are separated. For comma separated files, use `sep = ","`, for tab–delimited files, use `sep = "\t"`
`row.names`	A logical value (TRUE or FALSE) indicating whether or not save the rownames in the text file. (`row.names = FALSE` will not include row names)

For example, the following code will save the `pirates` dataframe as a tab–delimited text file called `pirates.txt` in my working directory:

```
# Write the pirates dataframe object to a tab-delimited
#  text file called pirates.txt in my working directory

write.table(x = pirates,
            file = "pirates.txt",  # Save the file as pirates.txt
            sep = "\t")            # Make the columns tab-delimited
```

If you want to save a file to a location outside of your working directory, just use the entire directory name. When you enter a long path name into the `file` argument of `write.table()`, R will look for that directory outside of your working directory. For example, to save a text file to my Desktop (which is outside of my working directory), I would set `file = "Users/nphillips/Desktop/pirates.txt"`.

```
# Write the pirates dataframe object to a tab-delimited
#  text file called pirates.txt to my desktop

write.table(x = pirates,
            file = "Users/nphillips/Desktop/pirates.txt",  # Save the file as pirates.txt to my desktop
            sep = "\t")                                     # Make the columns tab-delimited
```

9.6.2 read.table()

If you have a .txt file that you want to read into R, use the `read.table()` function.

Argument	Description
`file`	The document's file path relative to the working directory unless specified otherwise. For example `file = "mydata.txt"` looks for the text file directly in the working directory, while `file = "data/mydata.txt"` will look for the file in an existing folder called `data` inside the working directory.If the file is outside of your working directory, you can also specify a full file path (`file = "/Users/CaptainJack/Desktop/OctoberStudy/mydata.txt"`)
`header`	A logical value indicating whether the data has a header row – that is, whether the first row of the data represents the column names.
`sep`	A string indicating how the columns are separated. For comma separated files, use `sep = ","`, for tab–delimited files, use `sep = "\t"`
`stringsAsFactors`	A logical value indicating whether or not to convert strings to factors. I **always** set this to FALSE because I *hate*, *hate*, *hate* how R uses factors

The three critical arguments to `read.table()` are `file`, `sep`, `header` and `stringsAsFactors`. The `file` argument is a character value telling R where to find the file. If the file is in a folder in your working directory, just specify the path within your working directory (e.g.; `file = data/newdata.txt`. The `sep` argument tells R how the columns are separated in the file (again, for a comma–separated file, use `sep = ","`}, for a tab–delimited file, use `sep = "\t"`. The `header` argument is a logical value (TRUE or FALSE) telling R whether or not the first row in the data is the name of the data columns. Finally, the `stringsAsFactors` argument is a logical value indicating whether or not to convert strings to factors (I *always* set this to FALSE!)

Let's test this function out by reading in a text file titled `mydata.txt`. Since the text file is located a folder called `data` in my working directory, I'll use the file path `file = "data/mydata.txt"` and since the file is tab–delimited, I'll use the argument `sep = "\t"`:

```
# Read a tab-delimited text file called mydata.txt
#  from the data folder in my working directory into
#  R and store as a new object called mydata

mydata <- read.table(file = 'data/mydata.txt',    # file is in a data folder in my working directory
                     sep = '\t',                  # file is tab--delimited
                     header = TRUE,               # the first row of the data is a header row
                     stringsAsFactors = FALSE)    # do NOT convert strings to factors!!
```

9.6.3 Reading files directly from a web URL

A really neat feature of the `read.table()` function is that you can use it to load text files directly from the web (assuming you are online). To do this, just set the file path to the document's web URL (beginning with `http://`). For example, I have a text file stored at `http://goo.gl/jTNf6P`. You can import and save this tab–delimited text file as a new object called `fromweb` as follows:

```
# Read a text file from the web
fromweb <- read.table(file = 'http://goo.gl/jTNf6P',
                      sep = '\t',
                      header = TRUE)

# Print the result
fromweb
##           message randomdata
## 1 Congratulations!          1
## 2             you          2
## 3            just          3
## 4      downloaded          4
## 5            this          5
## 6           table          6
## 7            from          7
## 8             the          8
## 9            web!          9
```

I think this is pretty darn cool. This means you can save your main data files on Dropbox or a web-server, and always have access to it from any computer by accessing it from its web URL.

Debugging

When you run `read.table()`, you might receive an error like this:

Error in file(file, "rt") : cannot open the connection

In addition: Warning message:

In file(file, "rt") : cannot open file 'asdf': No such file or directory

If you receive this error, it's likely because you either spelled the file name incorrectly, or did not specify the correct directory location in the `file` argument.

9.7 Excel, SPSS, and other data files

A common question I hear is "How can I import an SPSS/Excel/... file into R?". The first answer to this question I always give is "You shouldn't". **S**hitty **P**iece of **S**hitty **S**hit files can contain information like variable descriptions that R doesn't know what to do with, and Excel files often contain something, like missing rows or cells with text instead of numbers, that can completely confuse R.

Rather than trying to import SPSS or Excel files directly in R, I always recommend first exporting/saving the original SPSS or Excel files as text `.txt.` files – both SPSS and Excel have options to do this. Then, once you have exported the data to a `.txt` file, you can read it into R using `read.table()`.

Warning: If you try to export an Excel file to a text file, it is a good idea to clean the file as much as you can first by, for example, deleting unnecessary columns, making sure that all numeric columns have numeric data, making sure the column names are simple (ie., single words without spaces or special

characters). If there is anything 'unclean' in the file, then R may still have problems reading it, even after you export it to a text file.

If you absolutely *have* to read a non-text file into R, check out the package called `foreign` (`install.packages("foreign")`). This package has functions for importing Stata, SAS and SPSS files directly into R. To read Excel files, try the package `xlsx` (`install.packages("xlsx")`). But again, in my experience it's *always* better to convert such files to simple text files first, and then read them into R with `read.table()`.

9.8 Additional tips

1. There are many functions other than `read.table()` for importing data. For example, the functions `read.csv` and `read.delim` are specific for importing comma-separated and tab-separated text files. In practice, these functions do the same thing as `read.table`, but they don't require you to specify a `sep` argument. Personally, I always use `read.table()` because it always works and I don't like trying to remember unnecessary functions.

9.9 Test your R Might!

1. In RStudio, open a new R Project in a new directory by clicking File – New Project. Call the directory `MyRProject`, and then select a directory on your computer for the project. This will be the project's working directory.
2. Outside of RStudio, navigate to the directory you selected in Question 1 and create three new folders – Call them `data`,R`, and`notes'.
3. Go back to RStudio and open a new R script. Save the script as `CreatingObjects.R` in the `R` folder you created in Question 2.
4. In the script, create new objects called `a`, `b`, and `c`. You can assign anything to these objects – from vectors to dataframes. If you can't think of any, use these:

```
a <- data.frame("sex" = c("m", "f", "m"),
                "age" = c(19, 43, 25),
                "favorite.movie" = c("Moon", "The Goonies", "Spice World"))
b <- mean(a$age)

c <- table(a$sex)
```

5. Send the code to the Console so the objects are stored in your current workspace. Use the `ls()` function to see that the objects are indeed stored in your workspace.
6. I have a tab–delimited text file called `club` at the following web address: http://nathanieldphillips.com/wp-content/uploads/2015/12/club.txt. Using `read.table()`, load the data as a new object called `club.df` in your workspace.
7. Using `write.table()`, save the dataframe as a tab–delimited text file called `club.txt` to the data folder you created in Question 2. Note: You won't use the text file again for this exercise, but now you have it handy in case you need to share it with someone who doesn't use R.
8. Save the three objects `a`, `b`, `c`, and `club.df` to an .RData file called "myobjects.RData" in your data folder using `save()`.
9. Clear your workspace using the `rm(list = ls())` function. Then, run the `ls()` function to make sure the objects are gone.

10. Open a new R script called `AnalyzingObjects.R` and save the script to the `R` folder you created in Question 2.
11. Now, in your `AnalyzingObjects.R` script, load the objects back into your workspace from the `myobjects.RData` file using the `load()` function. Again, run the `ls()` function to make sure all the objects are back in your workspace.
12. Add some R code to your `AnalyzingObjects.R` script. Calculate some means and percentages. Now save your `AnalyzingObjects.R` script, and then save all the objects in your workspace to `myobjects.RData`.
13. Congratulations! You are now a well-organized R Pirate! Quit RStudio and go outside for some relaxing pillaging.

Chapter 10

Advanced dataframe manipulation

In this chapter we'll cover some more advanced functions and procedures for manipulating dataframes.

```
# Exam data
exam <- data.frame(
  id = 1:5,
  q1 = c(1, 5, 2, 3, 2),
  q2 = c(8, 10, 9, 8, 7),
  q3 = c(3, 7, 4, 6, 4))

# Demographic data
demographics <- data.frame(
  id = 1:5,
  sex = c("f", "m", "f", "f", "m"),
  age = c(25, 22, 24, 19, 23))

# Combine exam and demographics
combined <- merge(x = exam,
              y = demographics,
              by = "id")

# Mean q1 score for each sex
aggregate(formula = q1 ~ sex,
          data = combined,
          FUN = mean)
##   sex  q1
## 1   f 2.0
```

Figure 10.1: Make your dataframes dance for you

```
## 2   m 3.5

# Median q3 score for each sex, but only for those
#   older than 20
aggregate(formula = q3 ~ sex,
          data = combined,
          subset = age > 20,
          FUN = mean)
##   sex  q3
## 1   f 3.5
## 2   m 5.5

# Many summary statistics by sex using dplyr!
library(dplyr)
combined %>% group_by(sex) %>%
  summarise(
    q1.mean = mean(q1),
    q2.mean = mean(q2),
    q3.mean = mean(q3),
    age.mean = mean(age),
    N = n())
## # A tibble: 2 x 6
##      sex q1.mean q2.mean q3.mean age.mean     N
##   <fctr>   <dbl>   <dbl>   <dbl>    <dbl> <int>
## 1      f     2.0     8.3     4.3       23     3
## 2      m     3.5     8.5     5.5       22     2
```

In Chapter 6, you learned how to calculate statistics on subsets of data using indexing. However, you may have noticed that indexing is not very intuitive and not terribly efficient. If you want to calculate statistics for many different subsets of data (e.g.; mean birth rate for every country), you'd have to write a new indexing command for each subset, which could take forever. Thankfully, R has some great built-in functions like `aggregate()` that allow you to easily apply functions (like `mean()`) to a dependent variable (like birth rate) for each level of one or more independent variables (like a country) with just a few lines of code.

10.1 `order()`: Sorting data

To sort the rows of a dataframe according to column values, use the `order()` function. The `order()` function takes one or more vectors as arguments, and returns an integer vector indicating the order of the vectors. You can use the output of `order()` to index a dataframe, and thus change its order.

Let's re-order the `pirates` data by height from the shortest to the tallest:

```
# Sort the pirates dataframe by height
pirates <- pirates[order(pirates$height),]

# Look at the first few rows and columns of the result
pirates[1:5, 1:4]
##      id    sex age height
## 39   39 female  25    130
## 854 854 female  25    130
## 30   30 female  26    135
## 223 223 female  28    135
```

```
## 351 351 female  36    137
```

By default, the `order()` function will sort values in ascending (increasing) order. If you want to order the values in descending (decreasing) order, just add the argument `decreasing = TRUE` to the `order()` function:

```
# Sort the pirates dataframe by height in decreasing order
pirates <- pirates[order(pirates$height, decreasing = TRUE),]

# Look at the first few rows and columns of the result
pirates[1:5, 1:4]
##      id  sex age height
## 2     2 male  31    209
## 793 793 male  25    209
## 430 430 male  26    201
## 292 292 male  29    201
## 895 895 male  27    201
```

To order a dataframe by several columns, just add additional arguments to `order()`. For example, to order the `pirates` by sex and then by height, we'd do the following:

```
# Sort the pirates dataframe by sex and then height
pirates <- pirates[order(pirates$sex, pirates$height),]
```

By default, the `order()` function will sort values in ascending (increasing) order. If you want to order the values in descending (decreasing) order, just add the argument `decreasing = TRUE` to the `order()` function:

```
# Sort the pirates dataframe by height in decreasing order
pirates <- pirates[order(pirates$height, decreasing = TRUE),]
```

10.2 merge(): Combining data

Argument	Description
`x, y`	Two dataframes to be merged
`by`	A string vector of 1 or more columns to match the data by. For example, `by = "id"` will combine columns that have matching values in a column called `"id"`. `by = c("last.name", "first.name")` will combine columns that have matching values in both `"last.name"` and `"first.name"`
`all`	A logical value indicating whether or not to include rows with non-matching values of `by`.

One of the most common data management tasks is **merging** (aka combining) two data sets together. For example, imagine you conduct a study where 5 participants are given a score from 1 to 5 on a risk assessment task. We can represent these data in a dataframe called `risk.survey`:

```
# Results from a risk survey
risk.survey <- data.frame(
  "participant" = c(1, 2, 3, 4, 5),
  "risk.score" = c(3, 4, 5, 3, 1))
```

Now, imagine that in a second study, you have participants complete a survey about their level of happiness

Table 10.2: Results from a survey on risk.

participant	risk.score
1	3
2	4
3	5
4	3
5	1

(on a scale of 0 to 100). We can represent these data in a new dataframe called `happiness.survey`:

```
happiness.survey <- data.frame(
  "participant" = c(4, 2, 5, 1, 3),
  "happiness.score" = c(20, 40, 50, 90, 53))
```

Now, we'd like to combine these data into one data frame so that the two survey scores for each participant are contained in one object. To do this, use `merge()`.

When you merge two dataframes, the result is a new dataframe that contains data from both dataframes. The key argument in `merge()` is `by`. The `by` argument specifies how rows should be matched during the merge. Usually, this will be something like an name, id number, or some other unique identifier.

Let's combine our risk and happiness survey using `merge()`. Because we want to match rows by the `participant.id` column, we'll specify `by = "participant.id"`. Additionally, because we want to include rows with potentially non-matching values, we'll include `all = TRUE`

```
# Combine the risk and happiness surveys by matching participant.id
combined.survey <- merge(x = risk.survey,
                         y = happiness.survey,
                         by = "participant",
                         all = TRUE)

# Print the result
combined.survey
##   participant risk.score happiness.score
## 1           1          3              90
## 2           2          4              40
## 3           3          5              53
## 4           4          3              20
## 5           5          1              50
```

For the rest of the chapter, we'll cover data aggregation functions. These functions allow you to quickly and easily calculate aggregated summary statistics over groups of data in a data frame. For example, you can use them to answer questions such as "What was the mean crew age for each ship?", or "What percentage of participants completed an attention check for each study condition?" We'll start by going over the `aggregate()` function.

10.3 aggregate(): Grouped aggregation

Argument	Description
`formula`	A formula in the form `y ~ x1 + x2 + ...` where y is the dependent variable, and x1, x2... are the independent variables. For example, `salary ~ sex + age` will aggregate a `salary` column at every unique combination of `sex` and `age`
`FUN`	A function that you want to apply to y at every level of the independent variables. E.g.; `mean`, or `max`.
`data`	The dataframe containing the variables in `formula`
`subset`	A subset of data to analyze. For example, `subset(sex == "f" & age > 20)` would restrict the analysis to females older than 20. You can ignore this argument to use all data.

The first aggregation function we'll cover is `aggregate()`. Aggregate allows you to easily answer questions in the form: "What is the value of the function `FUN` applied to a dependent variable `dv` at each level of one (or more) independent variable(s) `iv`?

```
# General structure of aggregate()
aggregate(formula = dv ~ iv, # dv is the data, iv is the group
          FUN = fun, # The function you want to apply
          data = df) # The dataframe object containing dv and iv
```

Let's give `aggregate()` a whirl. No...not a whirl...we'll give it a spin. Definitely a spin. We'll use `aggregate()` on the `ChickWeight` dataset to answer the question "What is the mean weight for each diet?"

If we wanted to answer this question using basic R functions, we'd have to write a separate command for each supplement like this:

```
# The WRONG way to do grouped aggregation.
#  We should be using aggregate() instead!
mean(ChickWeight$weight[ChickWeight$Diet == 1])
## [1] 103
mean(ChickWeight$weight[ChickWeight$Diet == 2])
## [1] 123
mean(ChickWeight$weight[ChickWeight$Diet == 3])
## [1] 143
mean(ChickWeight$weight[ChickWeight$Diet == 4])
## [1] 135
```

If you are ever writing code like this, there is almost always a simpler way to do it. Let's replace this code with a much more elegant solution using `aggregate()`.For this question, we'll set the value of the dependent variable Y to `weight`, x1 to `Diet`, and FUN to `mean`

```
# Calculate the mean weight for each value of Diet
aggregate(formula = weight ~ Diet,  # DV is weight, IV is Diet
          FUN = mean,               # Calculate the mean of each group
          data = ChickWeight)       # dataframe is ChickWeight
##   Diet weight
## 1    1    103
## 2    2    123
## 3    3    143
## 4    4    135
```

As you can see, the `aggregate()` function has returned a dataframe with a column for the independent variable `Diet`, and a column for the results of the function `mean` applied to each level of the independent variable. The result of this function is the same thing we'd got from manually indexing each level of `Diet` individually – but of course, this code is much simpler and more elegant!

You can also include a `subset` argument within an `aggregate()` function to apply the function to subsets of the original data. For example, if I wanted to calculate the mean chicken weights for each diet, but only when the chicks are less than 10 weeks old, I would do the following:

```
# Calculate the mean weight for each value of Diet,
#  But only when chicks are less than 10 weeks old

aggregate(formula = weight ~ Diet,  # DV is weight, IV is Diet
          FUN = mean,               # Calculate the mean of each group
          subset = Time < 10,       # Only when Chicks are less than 10 weeks old
          data = ChickWeight)       # dataframe is ChickWeight
##   Diet weight
## 1    1     58
## 2    2     63
## 3    3     66
## 4    4     69
```

You can also include multiple independent variables in the formula argument to `aggregate()`. For example, let's use `aggregate()` to now get the mean weight of the chicks for all combinations of both `Diet` and `Time`, but now only for weeks 0, 2, and 4:

```
# Calculate the mean weight for each value of Diet and Time,
#  But only when chicks are 0, 2 or 4 weeks okd

aggregate(formula = weight ~ Diet + Time,  # DV is weight, IVs are Diet and Time
          FUN = mean,                      # Calculate the mean of each group
          subset = Time %in% c(0, 2, 4),   # Only when Chicks are 0, 2, and 4 weeks old
          data = ChickWeight)              # dataframe is ChickWeight
##    Diet Time weight
## 1     1    0     41
## 2     2    0     41
## 3     3    0     41
## 4     4    0     41
## 5     1    2     47
## 6     2    2     49
## 7     3    2     50
## 8     4    2     52
## 9     1    4     56
## 10    2    4     60
## 11    3    4     62
## 12    4    4     64
```

10.4 dplyr

The `dplyr` package is a relatively new R package that allows you to do all kinds of analyses quickly and easily. It is especially useful for creating tables of summary statistics across specific groups of data. In this section, we'll go over a very brief overview of how you can use dplyr to easily do grouped aggregation. Just to be clear - you can use dplyr to do everything the `aggregate()` function does and much more! However, this will be a very brief overview and I strongly recommend you look at the help menu for dplyr for additional descriptions and examples.

To use the dplyr package, you first need to install it with `install.packages()` and load it:

```
install.packages("dplyr")      # Install dplyr (only necessary once)
library("dplyr")               # Load dplyr
```

Programming with dplyr looks a lot different than programming in standard R. dplyr works by combining objects (dataframes and columns in dataframes), functions (mean, median, etc.), and **verbs** (special commands in `dplyr`). In between these commands is a new operator called the **pipe** which looks like this: `%>%`}. The pipe simply tells R that you want to continue executing some functions or verbs on the object you are working on. You can think about this pipe as meaning 'and then…'

To aggregate data with `dplyr`, your code will look something like the following code. In this example, assume that the dataframe you want to summarize is called `my.df`, the variable you want to group the data by independent variables `iv1, iv2`, and the columns you want to aggregate are called `col.a`, `col.b` and `col.c`

```
# Template for using dplyr
my.df %>%                  # Specify original dataframe
  filter(iv3 > 30) %>%     # Filter condition
  group_by(iv1, iv2) %>%   # Grouping variable(s)
  summarise(
    a = mean(col.a),       # calculate mean of column col.a in my.df
    b = sd(col.b),         # calculate sd of column col.b in my.df
    c = max(col.c))        # calculate max on column col.c in my.df, ...
```

When you use dplyr, you write code that sounds like: "The original dataframe is XXX, now filter the dataframe to only include rows that satisfy the conditions YYY, now group the data at each level of the variable(s) ZZZ, now summarize the data and calculate summary functions XXX…"

Let's start with an example: Let's create a dataframe of aggregated data from the `pirates` dataset. I'll filter the data to only include pirates who wear a headband. I'll group the data according to the columns `sex` and `college`. I'll then create several columns of different summary statistic of some data across each grouping. To create this aggregated data frame, I will use the new function `group_by` and the verb `summarise`. I will assign the result to a new dataframe called `pirates.agg`:

```
pirates.agg <- pirates %>%                   # Start with the pirates dataframe
               filter(headband == "yes") %>% # Only pirates that wear hb
               group_by(sex, college) %>%    # Group by these variables
               summarise(
                         age.mean = mean(age),      # Define first summary...
                         tat.med = median(tattoos), # you get the idea...
                         n = n()                    # How many are in each group?
               ) # End

# Print the result
pirates.agg
## # A tibble: 6 x 5
## # Groups:   sex [?]
##      sex college age.mean tat.med     n
##    <chr>   <chr>    <dbl>   <dbl> <int>
## 1 female    CCCC       26      10   206
## 2 female   JSSFP       34      10   203
## 3   male    CCCC       23      10   358
## 4   male   JSSFP       32      10    85
## 5  other    CCCC       25      10    24
## 6  other   JSSFP       32      12    11
```

As you can see from the output on the right, our final object `pirates.agg` is the aggregated dataframe we want which aggregates all the columns we wanted for each combination of `sex` and `college` One key new function here is `n()`. This function is specific to dplyr and returns a frequency of values in a summary command.

Let's do a more complex example where we combine multiple verbs into one chunk of code. We'll aggregate data from the movies dataframe.

```
movies %>% # From the movies dataframe...
    filter(genre != "Horror" & time > 50) %>% # Select only these rows
    group_by(rating, sequel) %>% # Group by rating and sequel
    summarise( #
      frequency = n(), # How many movies in each group?
      budget.mean = mean(budget, na.rm = T),  # Mean budget?
      revenue.mean = mean(revenue.all), # Mean revenue?
      billion.p = mean(revenue.all > 1000)) # Percent of movies with revenue > 1000?
## # A tibble: 14 x 6
## # Groups:   rating [?]
##       rating sequel frequency budget.mean revenue.mean billion.p
##        <chr>  <int>     <int>       <dbl>        <dbl>     <dbl>
##  1         G      0        59       41.23          234    0.0000
##  2         G      1        12       92.92          357    0.0833
##  3     NC-17      0         2        3.75           18    0.0000
##  4 Not Rated      0        84        1.74           56    0.0000
##  5 Not Rated      1        12        0.67           66    0.0000
##  6        PG      0       312       51.78          191    0.0096
##  7        PG      1        62       77.21          372    0.0161
##  8     PG-13      0       645       52.09          168    0.0062
##  9     PG-13      1       120      124.16          524    0.1167
## 10         R      0       623       31.38          109    0.0000
## 11         R      1        42       58.25          226    0.0000
## 12      <NA>      0        86        1.65           34    0.0000
## 13      <NA>      1        15        5.51           48    0.0000
## 14      <NA>     NA        11        0.00           34    0.0000
```

As you can see, our result is a dataframe with 14 rows and 6 columns. The data are summarized from the movie dataframe, only include values where the genre is *not* Horror and the movie length is longer than 50 minutes, is grouped by rating and sequel, and shows several summary statistics.

10.4.1 Additional dplyr help

We've only scratched the surface of what you can do with `dplyr`. In fact, you can perform almost all of your R tasks, from loading, to managing, to saving data, in the `dplyr` framework. For more tips on using dplyr, check out the dplyr vignette at https://cran.r-project.org/web/packages/dplyr/vignettes/introduction.html. Or open it in RStudio by running the following command:

```
# Open the dplyr introduction in R
vignette("introduction", package = "dplyr")
```

There is also a very nice YouTube video covering `dplyr` at https://goo.gl/UY2AE1. Finally, consider also reading R for Data Science written by Garrett Grolemund and Hadley Wickham, which teaches R from the ground-up using the dplyr framework.

10.5 Additional aggregation functions

There are many, many other aggregation functions that I haven't covered in this chapter – mainly because I rarely use them. In fact, that's a good reminder of a peculiarity about R, there are many methods to

Table 10.4: Scores from an exam.

q1	q2	q3	q4	q5
1	1	1	1	1
0	0	0	1	0
0	1	1	1	0
0	1	0	1	1
0	0	0	1	1

achieve the same result, and your choice of which method to use will often come down to which method you just like the most.

10.5.1 rowMeans(), colMeans()

To easily calculate means (or sums) across all rows or columns in a matrix or dataframe, use `rowMeans()`, `colMeans()`, `rowSums()` or `colSums()`.

For example, imagine we have the following data frame representing scores from a quiz with 5 questions, where each row represents a student, and each column represents a question. Each value can be either 1 (correct) or 0 (incorrect)

```
# Some exam scores
exam <- data.frame("q1" = c(1, 0, 0, 0, 0),
                   "q2" = c(1, 0, 1, 1, 0),
                   "q3" = c(1, 0, 1, 0, 0),
                   "q4" = c(1, 1, 1, 1, 1),
                   "q5" = c(1, 0, 0, 1, 1))
```

Let's use `rowMeans()` to get the average scores for each student:

```
# What percent did each student get correct?
rowMeans(exam)
## [1] 1.0 0.2 0.6 0.6 0.4
```

Now let's use `colMeans()` to get the average scores for each *question*:

```
# What percent of students got each question correct?
colMeans(exam)
##  q1  q2  q3  q4  q5
## 0.2 0.6 0.4 1.0 0.6
```

Warning `rowMeans()` and `colMeans()` only work on numeric columns. If you try to apply them to non-numeric data, you'll receive an error.

10.5.2 apply family

There is an entire class of `apply` functions in R that apply functions to groups of data. For example, `tapply()`, `sapply()` and `lapply()` each work very similarly to `aggregate()`. For example, you can calculate the average length of movies by genre with `tapply()` as follows.

```
with(movies, tapply(X = time,        # DV is time
                    INDEX = genre,   # IV is genre
                    FUN = mean,      # function is mean
                    na.rm = TRUE))   # Ignore missing
```

```
##              Action           Adventure        Black Comedy
##                 113                 106                 113
##              Comedy Concert/Performance         Documentary
##                  99                  78                  69
##               Drama              Horror     Multiple Genres
##                 116                  99                 114
##             Musical             Reality     Romantic Comedy
##                 113                  44                 107
##   Thriller/Suspense             Western
##                 112                 121
```

`tapply()`, `sapply()`, and `lapply()` all work very similarly, their main difference is in the structure of their output. For example, `lapply()` returns a **list** (we'll cover lists in a future chapter).

10.6 Test your R might!: Mmmmm...caffeine

You're in charge of analyzing the results of an experiment testing the effects of different forms of caffeine on a measure of performance. In the experiment, 100 participants were given either Green tea or coffee, in doses of either 1 or 5 servings. They then performed a cognitive test where higher scores indicate better performance.

The data are stored in a tab–delimited dataframe at the following link: https://raw.githubusercontent.com/ndphillips/ThePiratesGuideToR/master/data/caffeinestudy.txt

1. Load the dataset from https://raw.githubusercontent.com/ndphillips/ThePiratesGuideToR/master/data/caffeinestudy.txt as a new dataframe called `caffeine`.
2. Calculate the mean age for each gender
3. Calculate the mean age for each drink
4. Calculate the mean age for each combined level of both gender and drink
5. Calculate the median score for each age

6. For men only, calculate the maximum score for each age
7. Create a dataframe showing, for each level of drink, the mean, median, maximum, and standard deviation of scores.
8. Only for females above the age of 20, create a table showing, for each combined level of drink and cups, the mean, median, maximum, and standard deviation of scores. Also include a column showing how many people were in each group.

Chapter 11

Plotting (I)

Sammy Davis Jr. was one of the greatest American performers of all time. If you don't know him already, Sammy was an American entertainer who lived from 1925 to 1990. The range of his talents was just incredible. He could sing, dance, act, and play multiple instruments with ease. So how is R like Sammy Davis Jr? Like Sammy Davis Jr., R is incredibly good at doing many different things. R does data analysis like Sammy dances, and creates plot like Sammy sings. If Sammy and R did just one of these things, they'd be great. The fact that they can do both is pretty amazing.

When you evaluate plotting functions in R, R can build the plot in different locations. The default location for plots is in a temporary plotting window within your R programming environment. In RStudio, plots will show up in the Plot window (typically on the bottom right hand window pane). In Base R, plots will show up in a Quartz window.

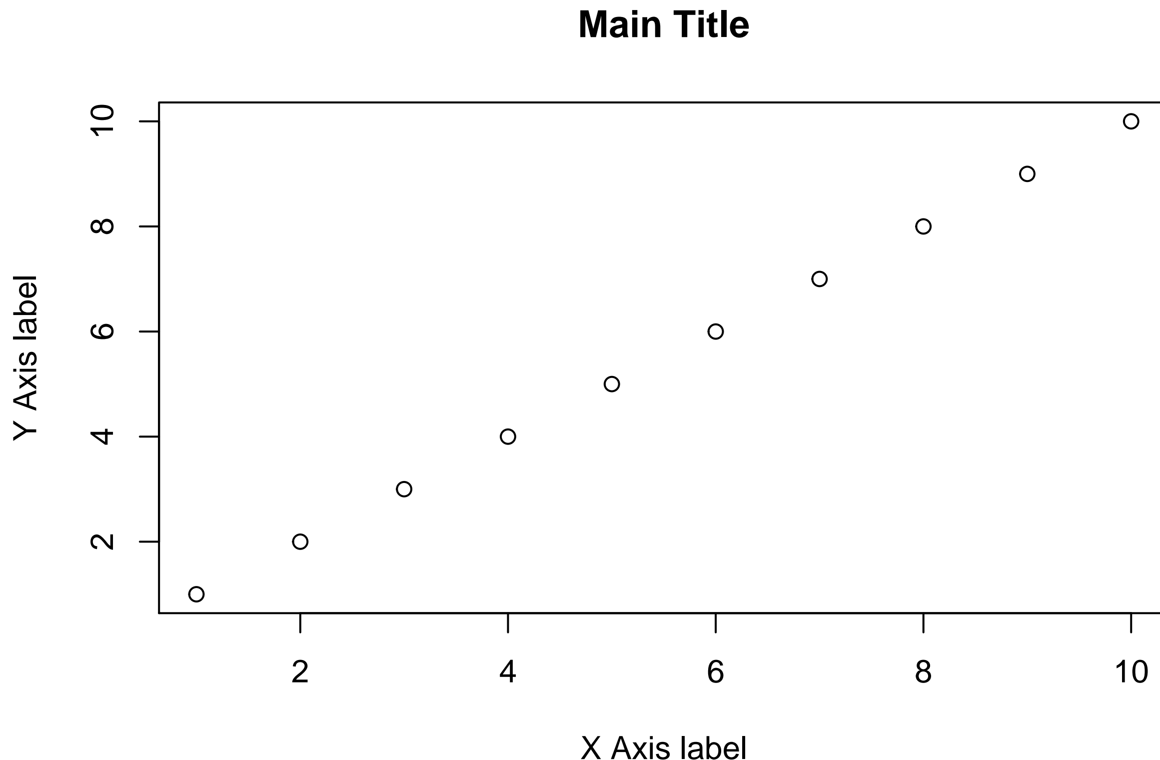
You can think of these plotting locations as canvases. You only have one canvas active at any given time, and any plotting command you run will put more plotting elements on your active canvas. Certain high–level plotting functions like `plot()` and `hist()` create brand new canvases, while other low–level plotting functions like `points()` and `segments()` place elements on top of existing canvases.

Don't worry if that's confusing for now – we'll go over all the details soon.

Let's start by looking at a basic scatterplot in R using the `plot()` function. When you execute the following code, you should see a plot open in a new window:

```
# A basic scatterplot
plot(x = 1:10,
     y = 1:10,
     xlab = "X Axis label",
     ylab = "Y Axis label",
     main = "Main Title")
```

Figure 11.1: The great Sammy Davis Jr. Do yourself a favor and spend an evening watching videos of him performing on YouTube. Image used entirely without permission.



Let's take a look at the result. We see an x–axis, a y–axis, 10 data points, an x–axis label, a y–axis label, and a main plot title. Some of these items, like the labels and data points, were entered as arguments to the function. For example, the main arguments x and y are vectors indicating the x and y coordinates of the (in this case, 10) data points. The arguments `xlab`, `ylab`, and `main` set the labels to the plot. However, there were many elements that I did not specify – from the x and y axis limits, to the color of the plotting points. As you'll discover later, you can change all of these elements (and many, many more) by specifying additional arguments to the `plot()` function. However, because I did not specify them, R used **default** values – values that R uses unless you tell it to use something else.

For the rest of this chapter, we'll go over the main plotting functions, along with the most common arguments you can use to customize the look of your plot.

11.1 Colors

Most plotting functions have a color argument (usually `col`) that allows you to specify the color of whatever your plotting. There are many ways to specify colors in R, let's start with the easiest ways.

11.1.1 Colors by name

The easiest way to specify a color is to enter its name as a string. For example `col = "red"` is R's default version of the color red. Of course, all the basic colors are there, but R also has tons of quirky colors like `"snow"`, `"papayawhip"` and `"lawngreen"`. Figure 11.2 shows 100 randomly selected named colors.

To see all 657 color names in R, run the code `colors()`. Or to see an interactive demo of colors, run `demo("colors")`.

Figure 11.2: 100 random named colors (out of all 657) in R.

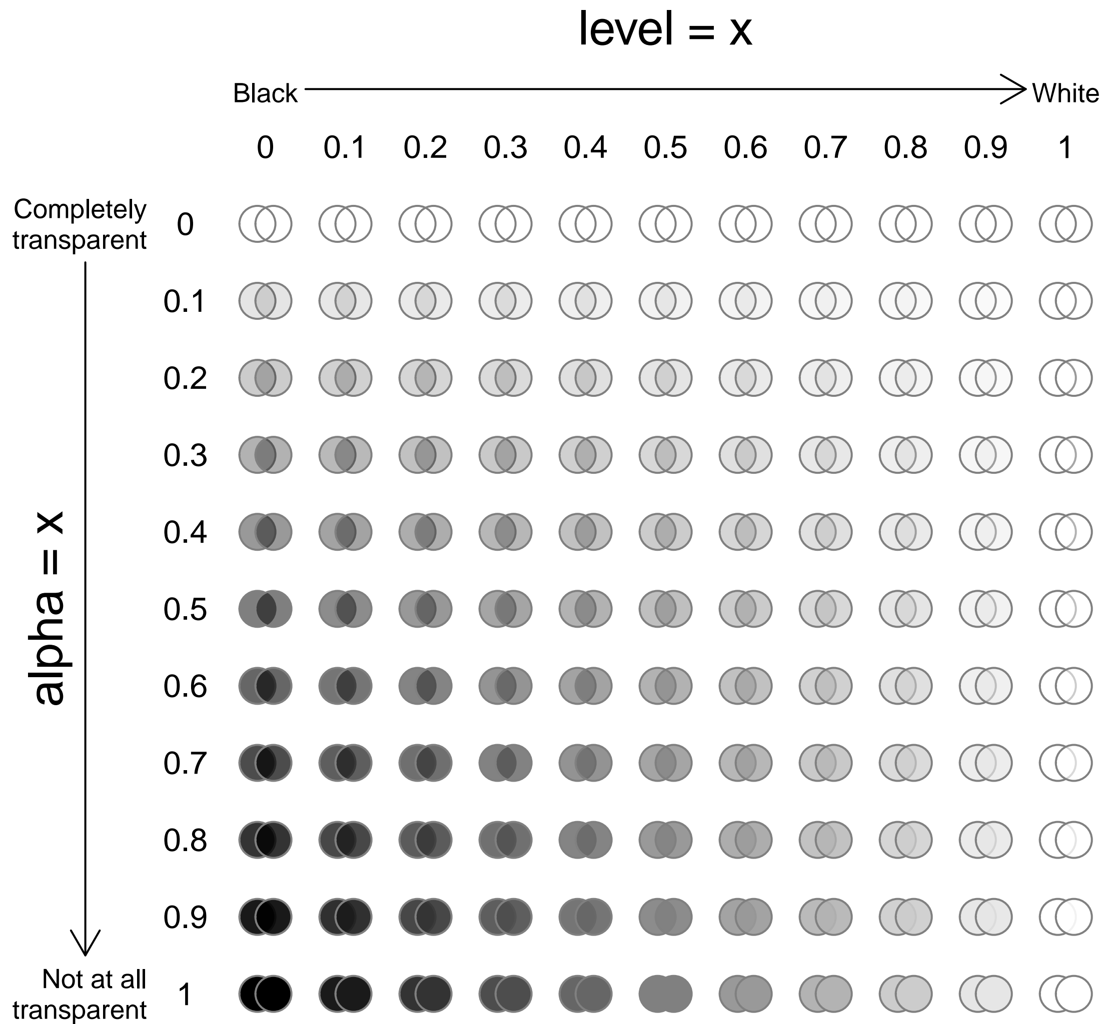


Figure 11.3: Examples of gray(level, alpha)

11.1.2 gray()

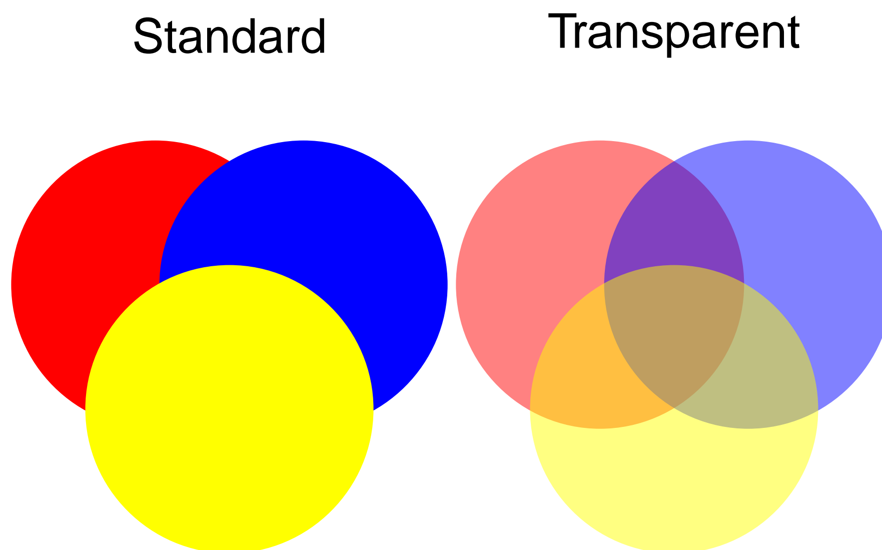
Table 11.1: `gray()` function arguments

Argument	Description
`level`	Lightness: `level = 1` = totally white, `level = 0` = totally black
`alpha`	Transparency: `alpha = 0` = totally transparent, `alpha = 1` = not transparent at all.

If you're into erotic romance and BDSM, then you might be interested in Shades of Gray. If so, the function `gray(x)` is your answer. The `gray()` function takes two arguments, `level` and `alpha`, and returns a shade of gray. For example, `gray(level = 1)` will return white. The second `alpha` argument specifies how transparent to make the color on a scale from 0 (completely transparent), to 1 (not transparent at all). The default value for alpha is 1 (not transparent at all). See Figure 11.3 for examples.

11.1.3 yarrr::transparent()

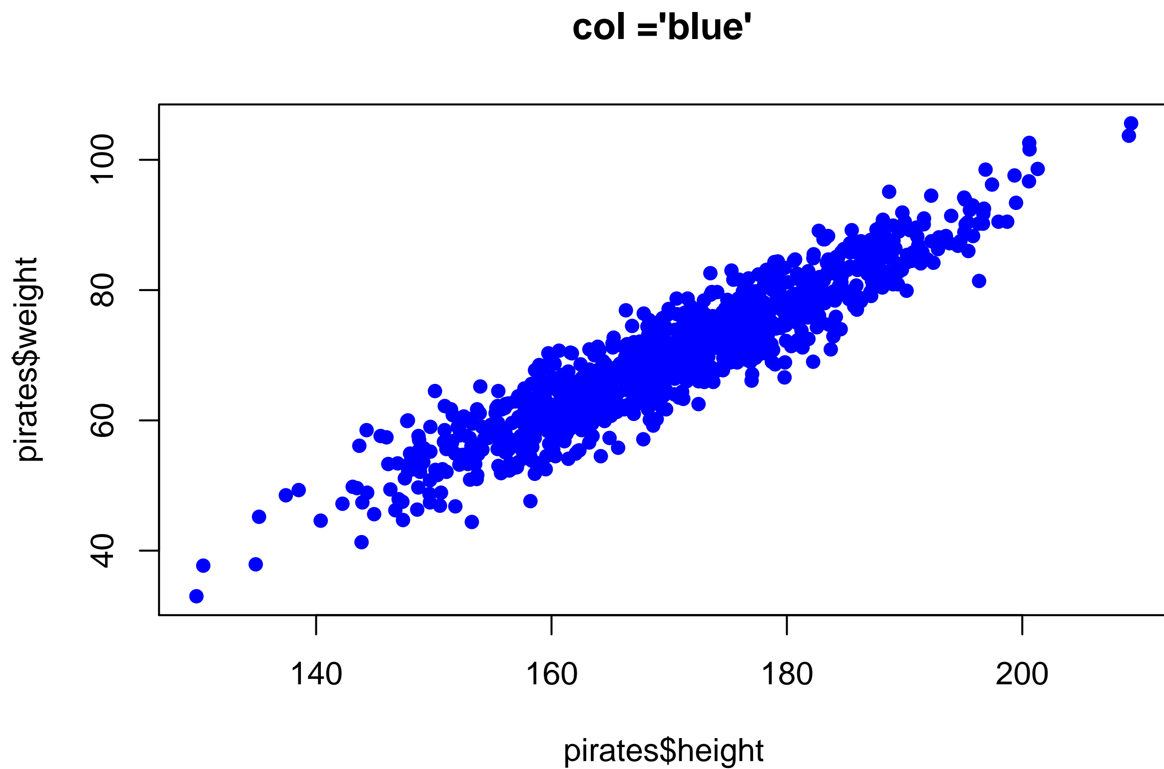
I don't know about you, but I almost always find transparent colors to be more appealing than solid colors. Not only do they help you see when multiple points are overlapping, but they're just much nicer to look at. Just look at the overlapping circles in the plot below.



Unfortunately, as far as I know, base-R does not make it easy to make transparent colors. Thankfully, there is a function in the `yarrr` package called `transparent` that makes it very easy to make any color transparent. To use it, just enter the original color as the main argument `orig.col`, then enter how transparent you want to make it (from 0 to 1) as the second argument `trans.val`.

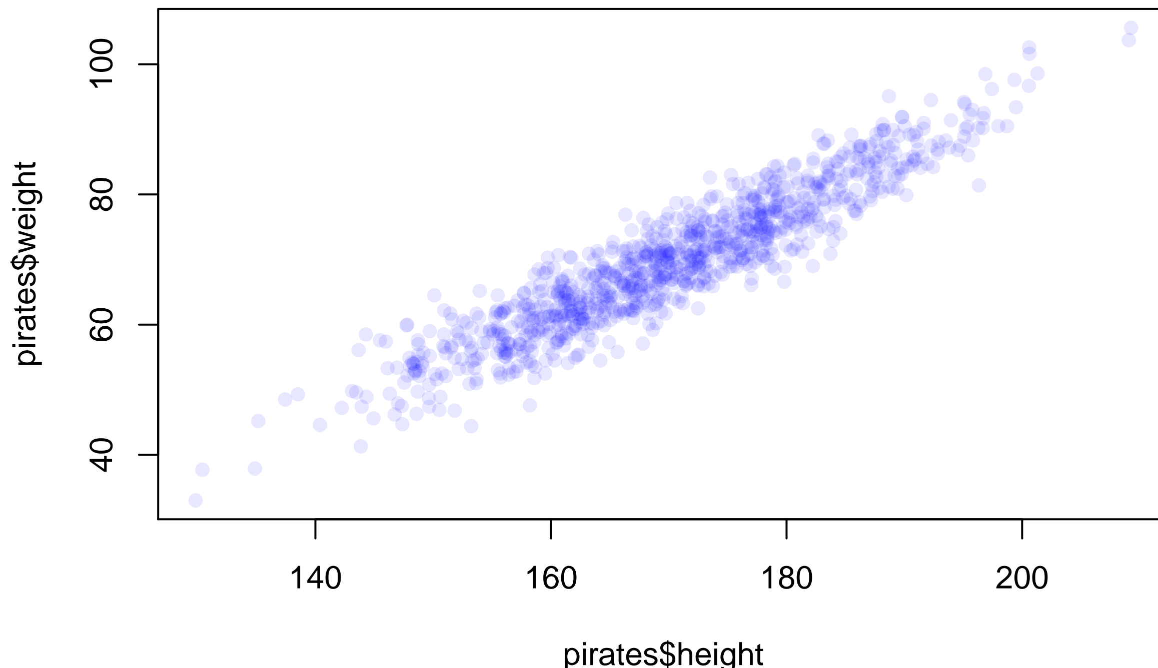
Here is a basic scatterplot with standard (non-transparent) colors:

```
# Plot with Standard Colors
plot(x = pirates$height,
     y = pirates$weight,
     col = "blue",
     pch = 16,
     main = "col ='blue'")
```



Now here's the same plot using the `transparent()` function in the `yarrr` package:

```
# Plot with transparent colors using the transparent() function in the yarrr package
plot(x = pirates$height,
     y = pirates$weight,
     col = yarrr::transparent("blue", trans.val = .9),
     pch = 16,
     main = "col = yarrr::transparent('blue', .9)")
```



Later on in the book, we'll cover more advanced ways to come up with colors using color palettes (using the RColorBrewer package or the `piratepal()` function in the yarrr package) and functions that generate shades of colors based on numeric data (like the `colorRamp2()` function in the `circlize` package).

11.2 Plotting arguments

Most plotting functions have *tons* of optional arguments (also called parameters) that you can use to customize virtually everything in a plot. To see all of them, look at the help menu for `par` by executing `?par`. However, the good news is that you don't need to specify all possible parameters you create a plot. Instead, there are only a few critical arguments that you must specify - usually one or two vectors of data. For any optional arguments that you do not specify, R will use either a default value, or choose a value that makes sense based on the data you specify.

In the following examples, I will to cover the main plotting parameters for each plotting type. However, the best way to learn what you can, and can't, do with plots, is to try to create them yourself!

I think the best way to learn how to create plots is to see some examples. Let's start with the main high-level plotting functions.

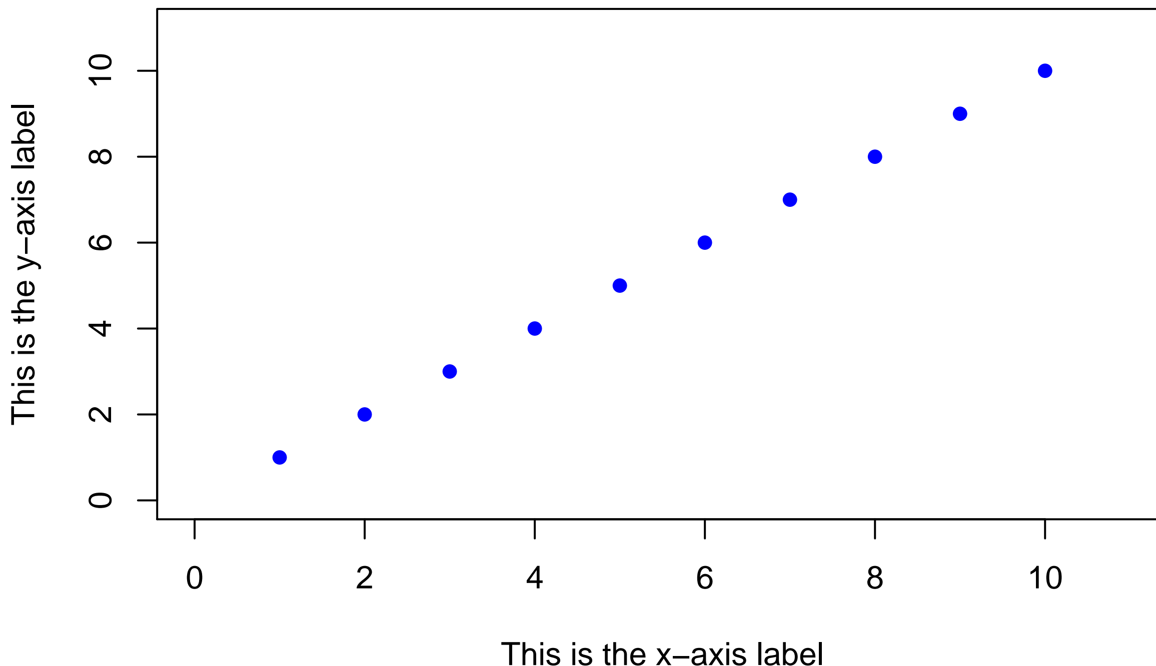
11.3 Scatterplot: `plot()`

The most common high-level plotting function is `plot(x, y)`. The `plot()` function makes a scatterplot from two vectors x and y, where the x vector indicates the x (horizontal) values of the points, and the y vector indicates the y (vertical) values.

Table 11.2: `plot()` function arguments

Argument	Description
`x, y`	Vectors of equal length specifying the x and y values of the points
`type`	Type of plot. `"l"` means lines, `"p"` means points, `"b"` means lines and points, `"n"` means no plotting
`main`, `xlab`, `ylab`	Strings giving labels for the plot title, and x and y axes
`xlim`, `ylim`	Limits to the axes. For example, `xlim = c(0, 100)` will set the minimum and maximum of the x-axis to 0 and 100.
`pch`	An integer indicating the type of plotting symbols (see `?points` and section below), or a string specifying symbols as text. For example, `pch = 21` will create a two-color circle, while `pch = "P"` will plot the character `"P"`. To see all the different symbol types, run `?points`.
`col`	Main color of the plotting symbols. For example `col = "red"` will create red symbols.
`cex`	A numeric vector specifying the size of the symbols (from 0 to Inf). The default size is 1. `cex = 4` will make the points very large, while `cex = .5` will make them very small.

```
plot(x = 1:10,                          # x-coordinates
     y = 1:10,                          # y-coordinates
     type = "p",                        # Just draw points (no lines)
     main = "My First Plot",
     xlab = "This is the x-axis label",
     ylab = "This is the y-axis label",
     xlim = c(0, 11),                   # Min and max values for x-axis
     ylim = c(0, 11),                   # Min and max values for y-axis
     col = "blue",                      # Color of the points
     pch = 16,                          # Type of symbol (16 means Filled circle)
     cex = 1)                            # Size of the symbols
```



Aside from the x and y arguments, all of the arguments are optional. If you don't specify a specific argument, then R will use a default value, or try to come up with a value that makes sense. For example, if you don't specify the `xlim` and `ylim` arguments, R will set the limits so that all the points fit inside the plot.

11.3.1 Symbol types: `pch`

When you create a plot with `plot()` (or points with `points()`), you can specify the type of symbol with the `pch` argument. You can specify the symbol type in one of two ways: with an integer, or with a string. If you use a string (like `"p"`), R will use that text as the plotting symbol. If you use an integer value, you'll get the symbol that correspond to that number. See Figure for all the symbol types you can specify with an integer.

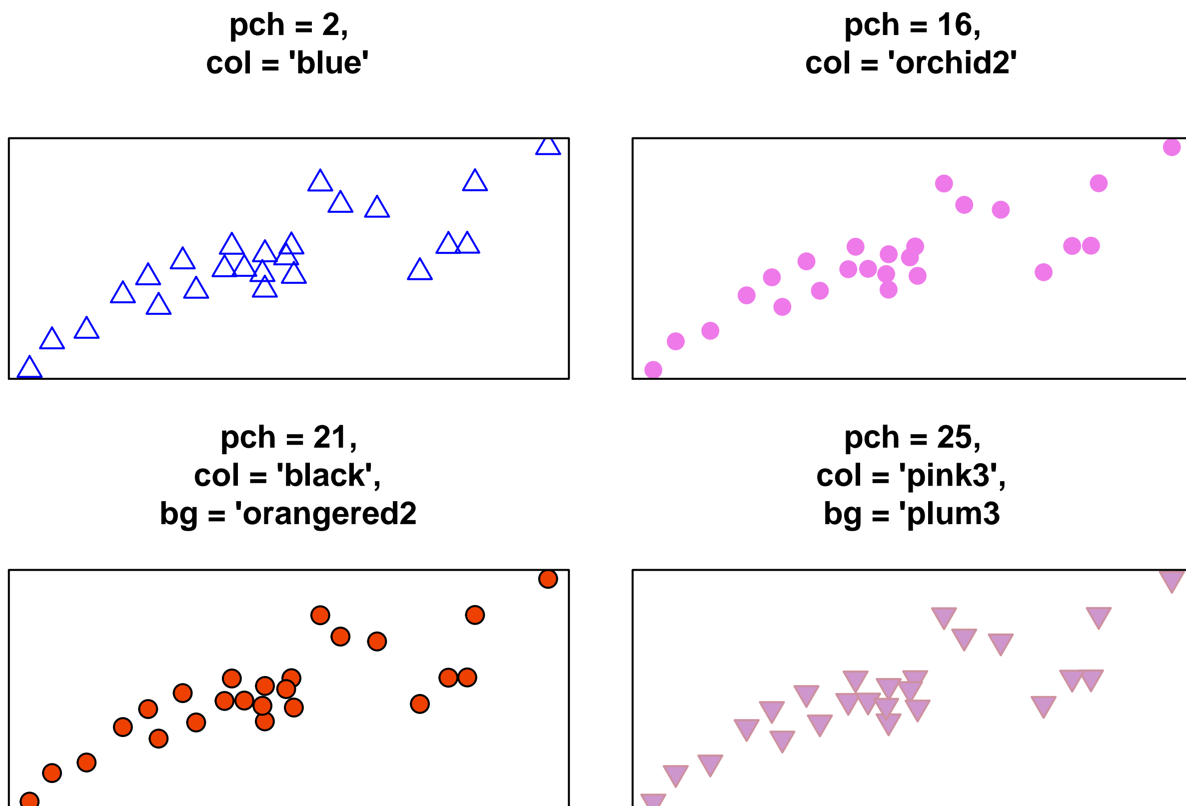
Symbols differ in their shape and how they are colored. Symbols 1 through 14 only have borders and are always empty, while symbols 15 through 20 don't have a border and are always filled. Symbols 21 through 25 have both a border and a filling. To specify the border color or background for symbols 1 through 20, use the `col` argument. For symbols 21 through 25, you set the color of the border with `col`, and the color of the background using `bg`

Let's look at some different symbol types in action when applied to the same data:

pch = _

1 ○ 6 ▽ 11 ✡ 16 ● 21 ○
2 △ 7 ⊠ 12 ⊞ 17 ▲ 22 □
3 + 8 ✳ 13 ⊗ 18 ◆ 23 ◇
4 × 9 ◈ 14 ⧅ 19 ● 24 △
5 ◇ 10 ⊕ 15 ■ 20 • 25 ▽

Figure 11.4: The symbol types associated with the pch plotting parameter.



11.4 Histogram: `hist()`

Table 11.3: `hist()` function arguments

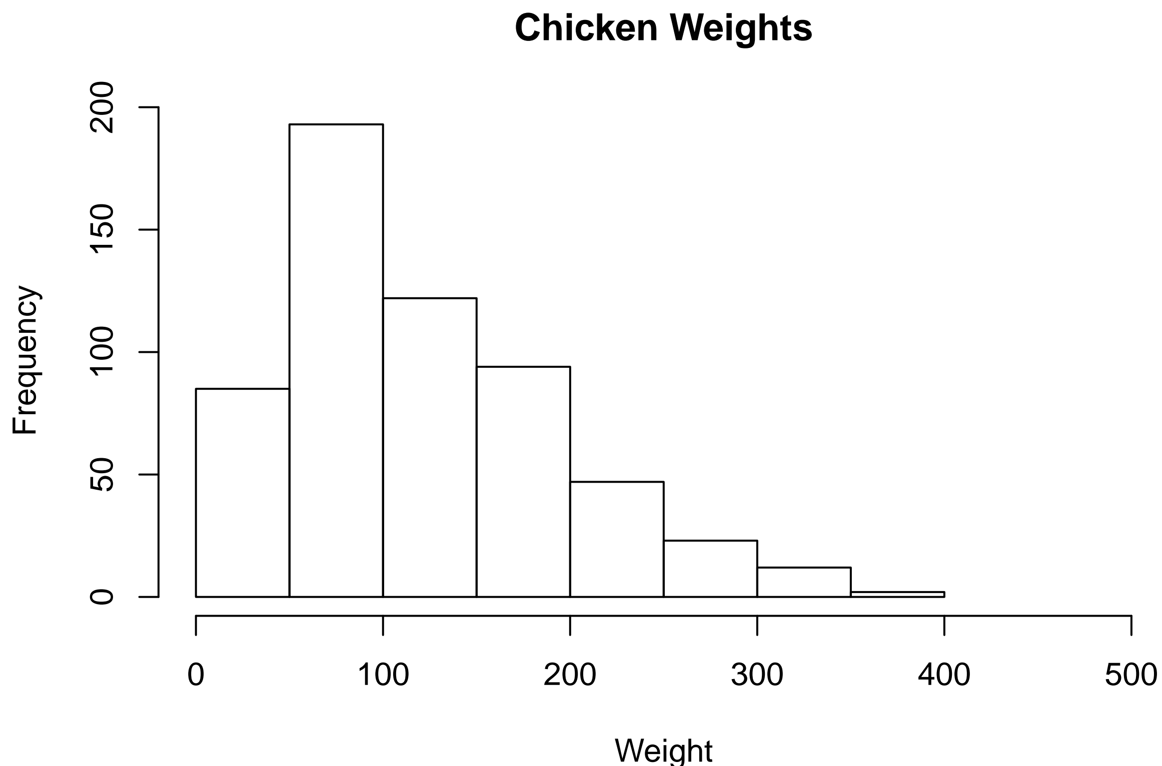
Argument	Description
`x`	Vector of values
`breaks`	How should the bin sizes be calculated? Can be specified in many ways (see `?hist` for details)

Argument	Description
`freq`	Should frequencies or probabilities be plotted? `freq = TRUE` shows frequencies, `freq = FALSE` shows probabilities.
`col`, `border`	Colors of the bin filling (`col`) and border (`border`)

Histograms are the most common way to plot a vector of numeric data. To create a histogram we'll use the `hist()` function. The main argument to `hist()` is a `x`, a vector of numeric data. If you want to specify how the histogram bins are created, you can use the `breaks` argument. To change the color of the border or background of the bins, use `col` and `border`:

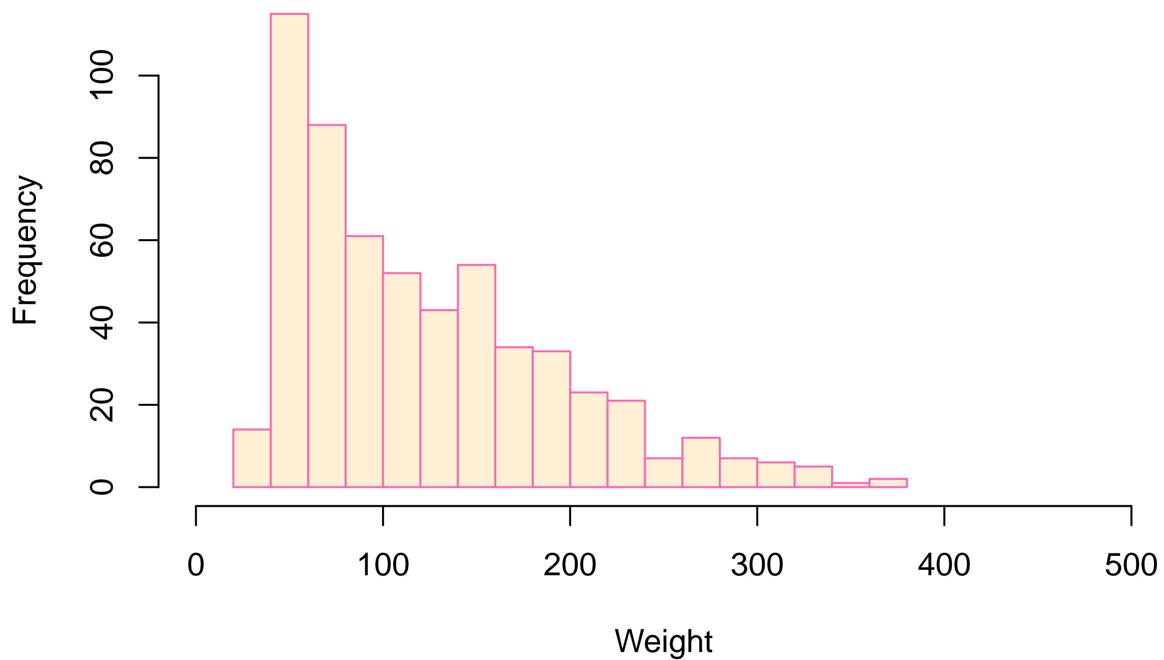
Let's create a histogram of the weights in the ChickWeight dataset:

```
hist(x = ChickWeight$weight,
     main = "Chicken Weights",
     xlab = "Weight",
     xlim = c(0, 500))
```



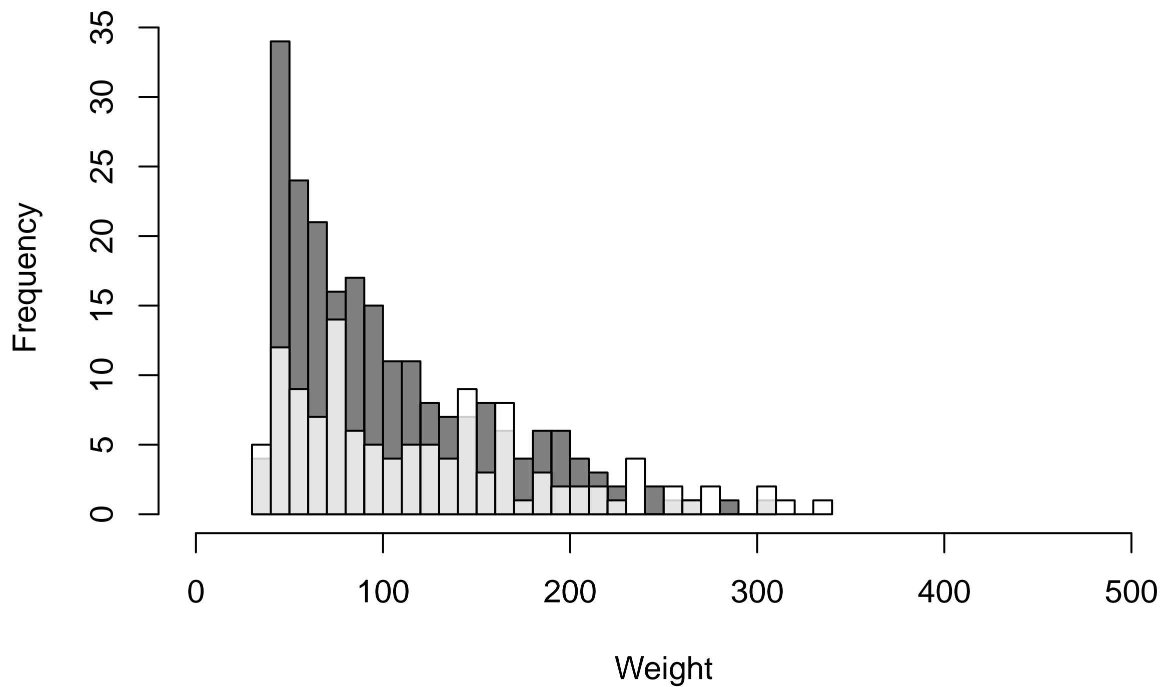
We can get more fancy by adding additional arguments like `breaks = 20` to force there to be 20 bins, and `col = "papayawhip"` and `bg = "hotpink"` to make it a bit more colorful:

```
hist(x = ChickWeight$weight,
     main = "Fancy Chicken Weight Histogram",
     xlab = "Weight",
     ylab = "Frequency",
     breaks = 20, # 20 Bins
     xlim = c(0, 500),
     col = "papayawhip", # Filling Color
     border = "hotpink") # Border Color
```



If you want to plot two histograms on the same plot, for example, to show the distributions of two different groups, you can use the `add = TRUE` argument to the second plot.

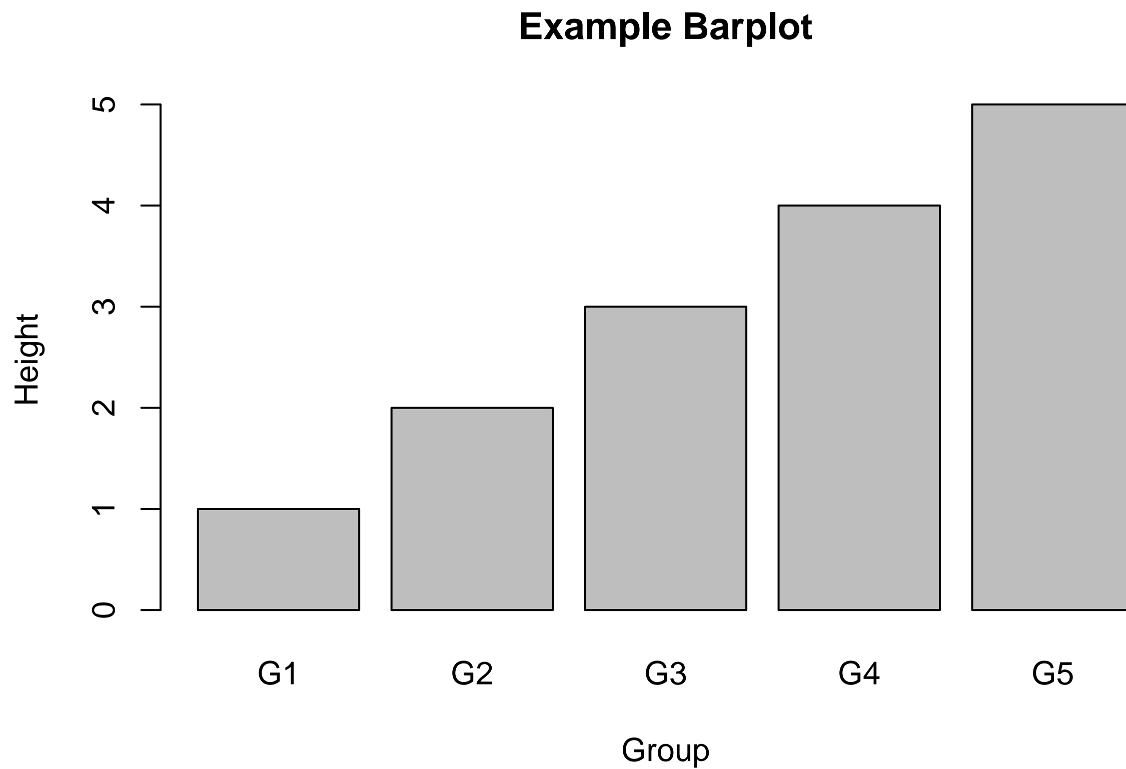
```
hist(x = ChickWeight$weight[ChickWeight$Diet == 1],
     main = "Two Histograms in one",
     xlab = "Weight",
     ylab = "Frequency",
     breaks = 20,
     xlim = c(0, 500),
     col = gray(0, .5))

hist(x = ChickWeight$weight[ChickWeight$Diet == 2],
     breaks = 30,
     add = TRUE, # Add plot to previous one!
     col = gray(1, .8))
```



11.5 Barplot: `barplot()`

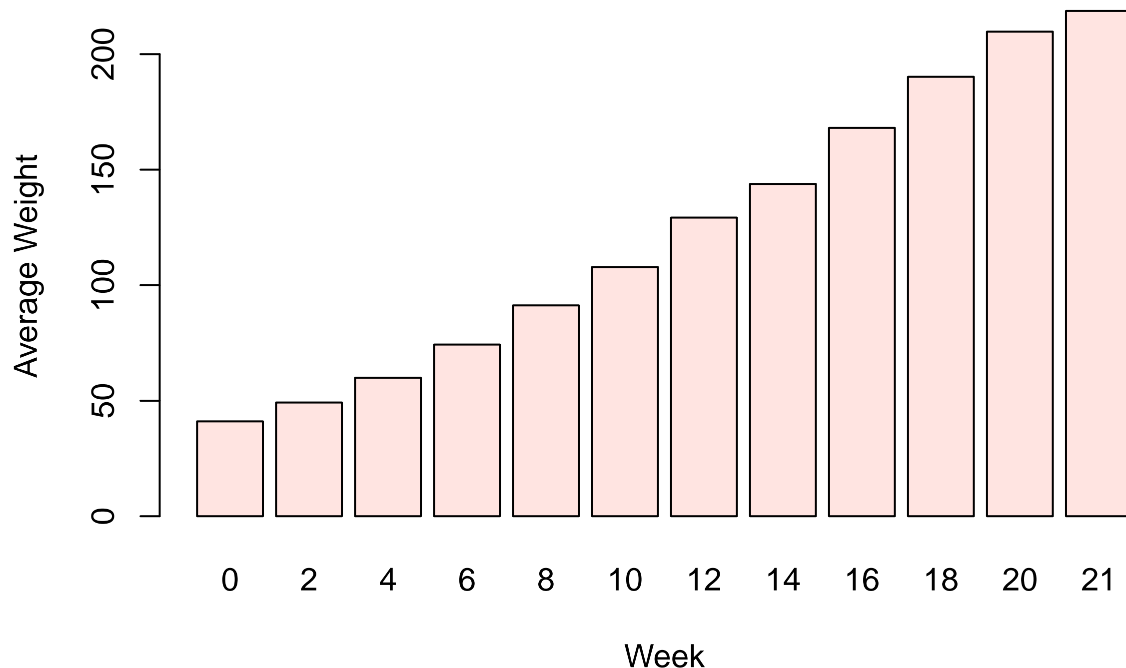
A barplot typically shows summary statistics for different groups. The primary argument to a barplot is `height`: a vector of numeric values which will generate the height of each bar. To add names below the bars, use the `names.arg` argument. For additional arguments specific to `barplot()`, look at the help menu with `?barplot`:

```
barplot(height = 1:5,  # A vector of heights
        names.arg = c("G1", "G2", "G3", "G4", "G5"), # A vector of names
        main = "Example Barplot",
        xlab = "Group",
        ylab = "Height")
```



Of course, you should plot more interesting data than just a vector of integers with a barplot. In the plot below, I create a barplot with the average weight of chickens for each week:

```
# Calculate mean weights for each time period
diet.weights <- aggregate(weight ~ Time,
                          data = ChickWeight,
                          FUN = mean)

# Create barplot
barplot(height = diet.weights$weight,
        names.arg = diet.weights$Time,
        xlab = "Week",
        ylab = "Average Weight",
        main = "Average Chicken Weights by Time",
        col = "mistyrose")
```



11.5.1 Clustered barplot

If you want to create a clustered barplot, with different bars for different groups of data, you can enter a matrix as the argument to `height`. R will then plot each column of the matrix as a separate set of bars. For example, let's say I conducted an experiment where I compared how fast pirates can swim under four conditions: Wearing clothes versus being naked, and while being chased by a shark versus not being chased by a shark. Let's say I conducted this experiment and calculated the following average swimming speed:

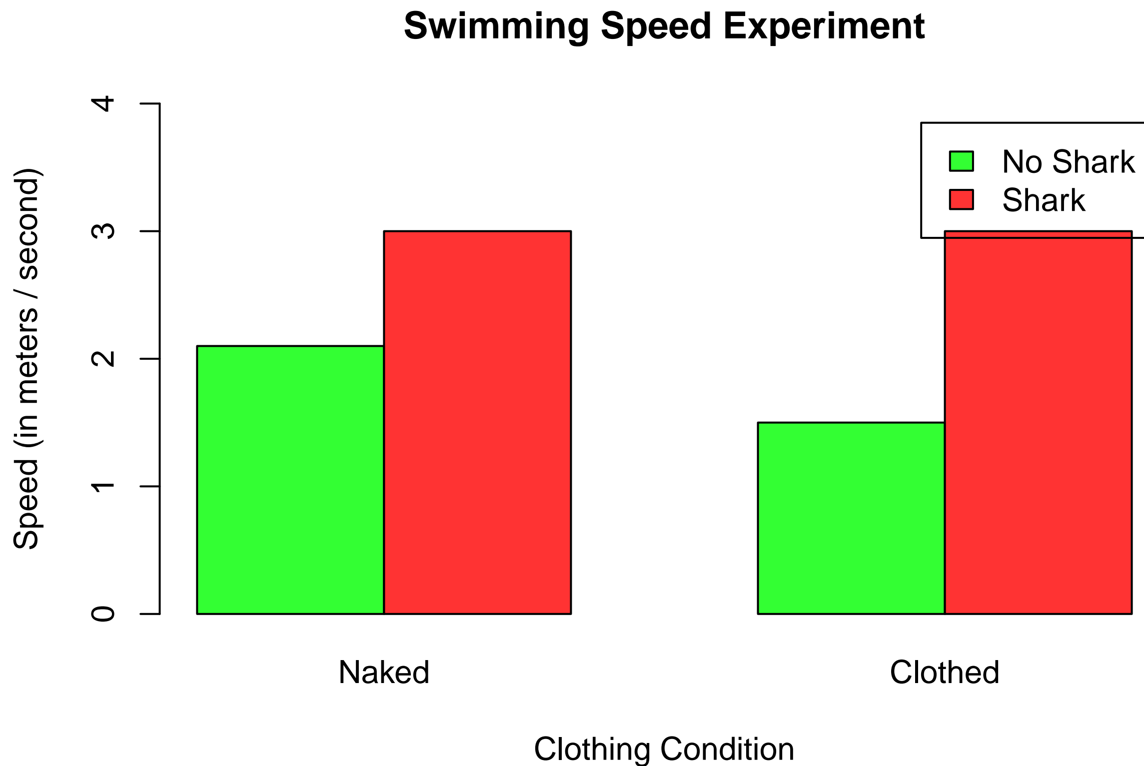
	Naked	Clothed
No Shark	2.1	1.5
Shark	3.0	3.0

I can represent these data in a matrix as follows. In order for the final barplot to include the condition names, I'll add row and column names to the matrix with `colnames()` and `rownames()`

```
swim.data <- cbind(c(2.1, 3), # Naked Times
                   c(1.5, 3)) # Clothed Times

colnames(swim.data) <- c("Naked", "Clothed")
rownames(swim.data) <- c("No Shark", "Shark")

# Print result
swim.data
##          Naked Clothed
## No Shark   2.1     1.5
## Shark      3.0     3.0
```

Now, when I enter this matrix as the `height = swim.data` argument to `barplot()`, I'll get multiple bars.

```
barplot(height = swim.data,
        beside = TRUE,                          # Put the bars next to each other
        legend.text = TRUE,                     # Add a legend
        col = c(transparent("green", .2),
                transparent("red", .2)),
        main = "Swimming Speed Experiment",
        ylab = "Speed (in meters / second)",
        xlab = "Clothing Condition",
        ylim = c(0, 4))
```



11.6 pirateplot()

Table 11.4: `pirateplot()` function arguments

Argument	Description
`formula`	A formula specifying a y-axis variable as a function of 1, 2 or 3 x-axis variables. For example, `formula = weight ~ Diet + Time` will plot `weight` as a function of `Diet` and `Time`
`data`	A dataframe containing the variables specified in `formula`
`theme`	A plotting theme, can be an integer from 1 to 4. Setting `theme = 0` will turn off all plotting elements so you can then turn them on individually.
`pal`	The color palette. Can either be a named color palette from the `piratepal()` function (e.g. `"basel"`, `"xmen"`, `"google"`) or a standard R color. For example, make a black and white plot, set `pal = "black"`
`cap.beans`	If `cap.beans = TRUE`, beans will be cut off at the maximum and minimum data values

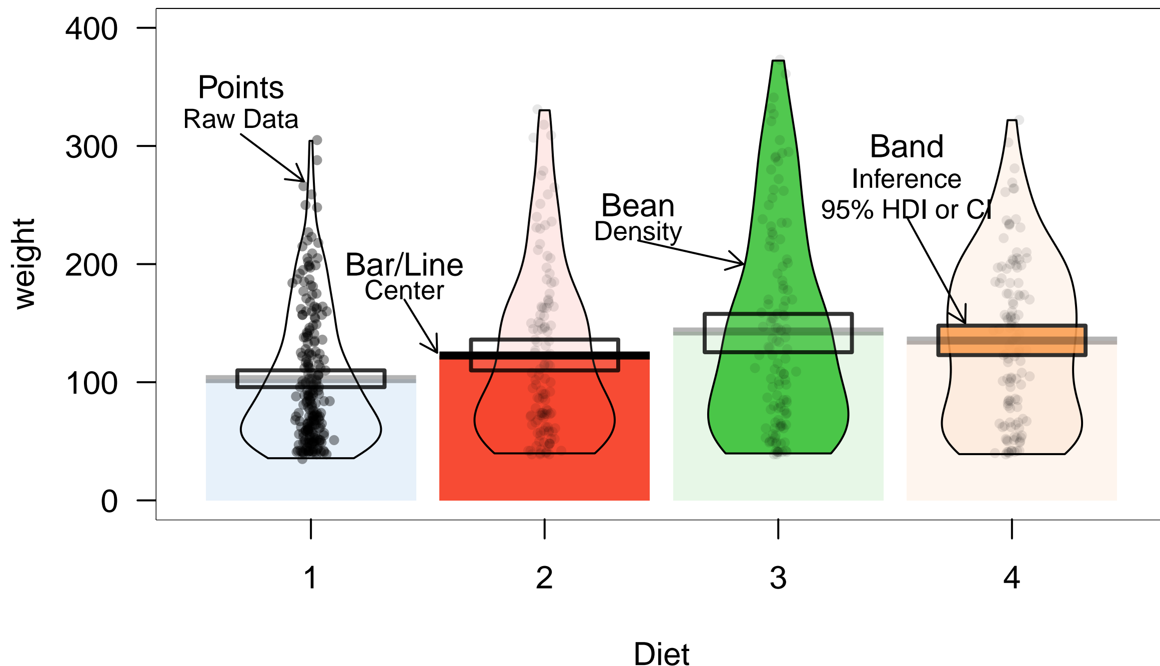


Figure 11.5: The pirateplot(), an R pirate's favorite plot!

A pirateplot a plot contained in the `yarrr` package written specifically by, and for R pirates The pirateplot is an easy-to-use function that, unlike barplots and boxplots, can easily show raw data, descriptive statistics, and inferential statistics in one plot. Figure 11.5 shows the four key elements in a pirateplot:

Table 11.5: 4 elements of a `pirateplot()`

Element	Description
Points	**Raw** data.
Bar / Line	**Descriptive** statistic, usually the mean or median
Bean	Smoothed density curve showing the full data distribution.
Band	**Inference** around the mean, either a Bayesian Highest Density Interval (HDI), or a Confidence Interval (CI)

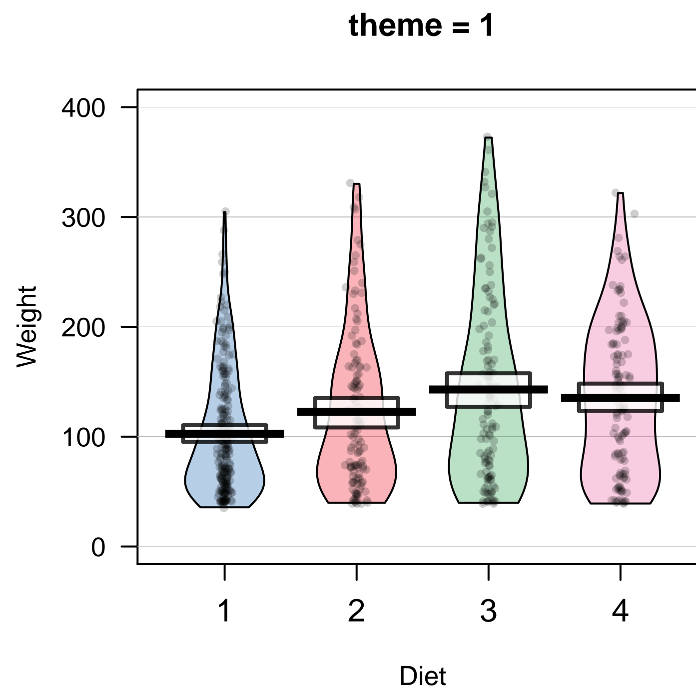
The two main arguments to `pirateplot()` are `formula` and `data`. In `formula`, you specify plotting variables in the form `y ~ x`, where `y` is the name of the dependent variable, and `x` is the name of the independent variable. In `data`, you specify the name of the dataframe object where the variables are stored.

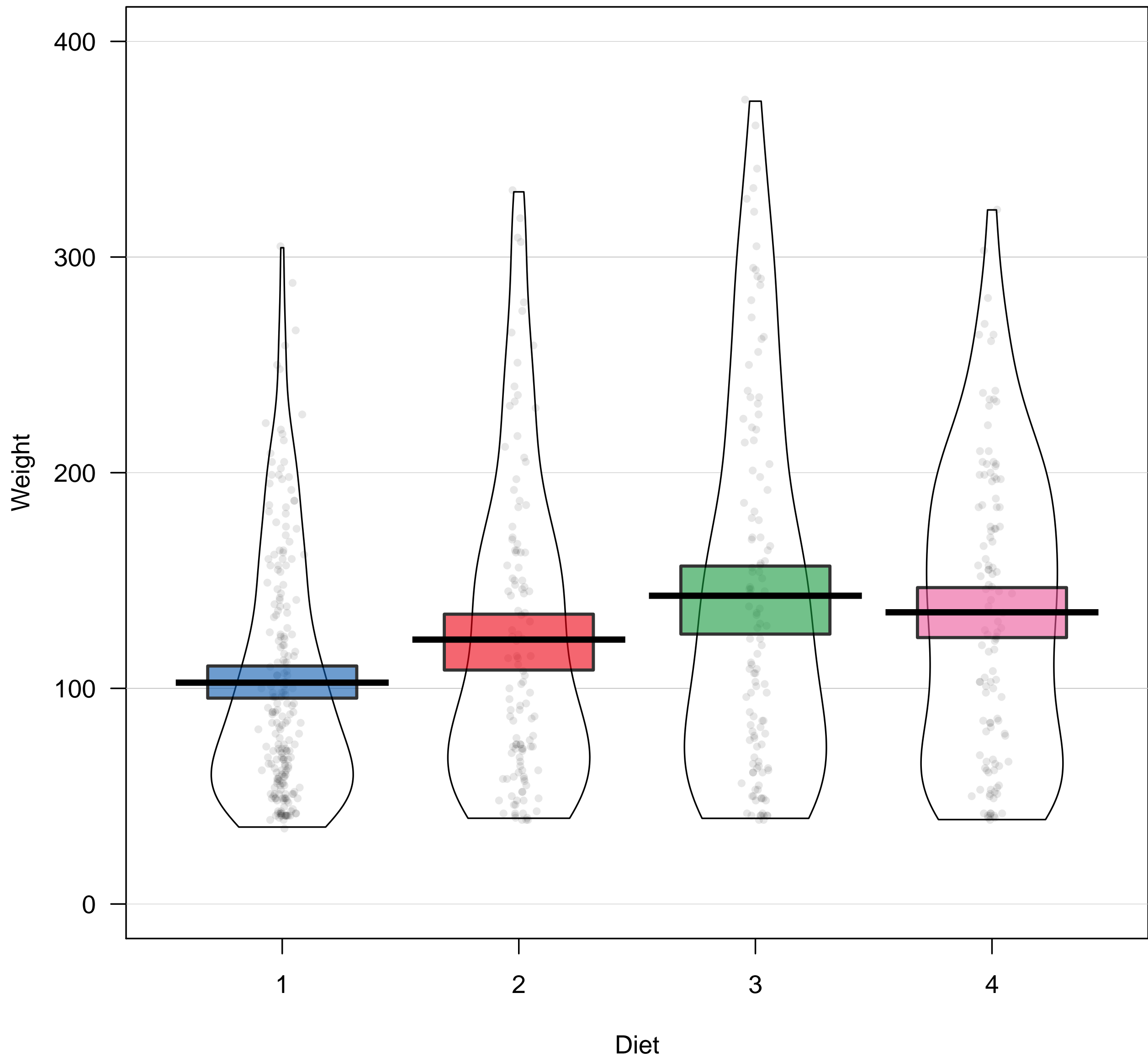
Let's create a pirateplot of the ChickWeight data. I'll set the dependent variable to `weight`, and the independent variable to `Time` using the argument `formula = weight ~ Time`:

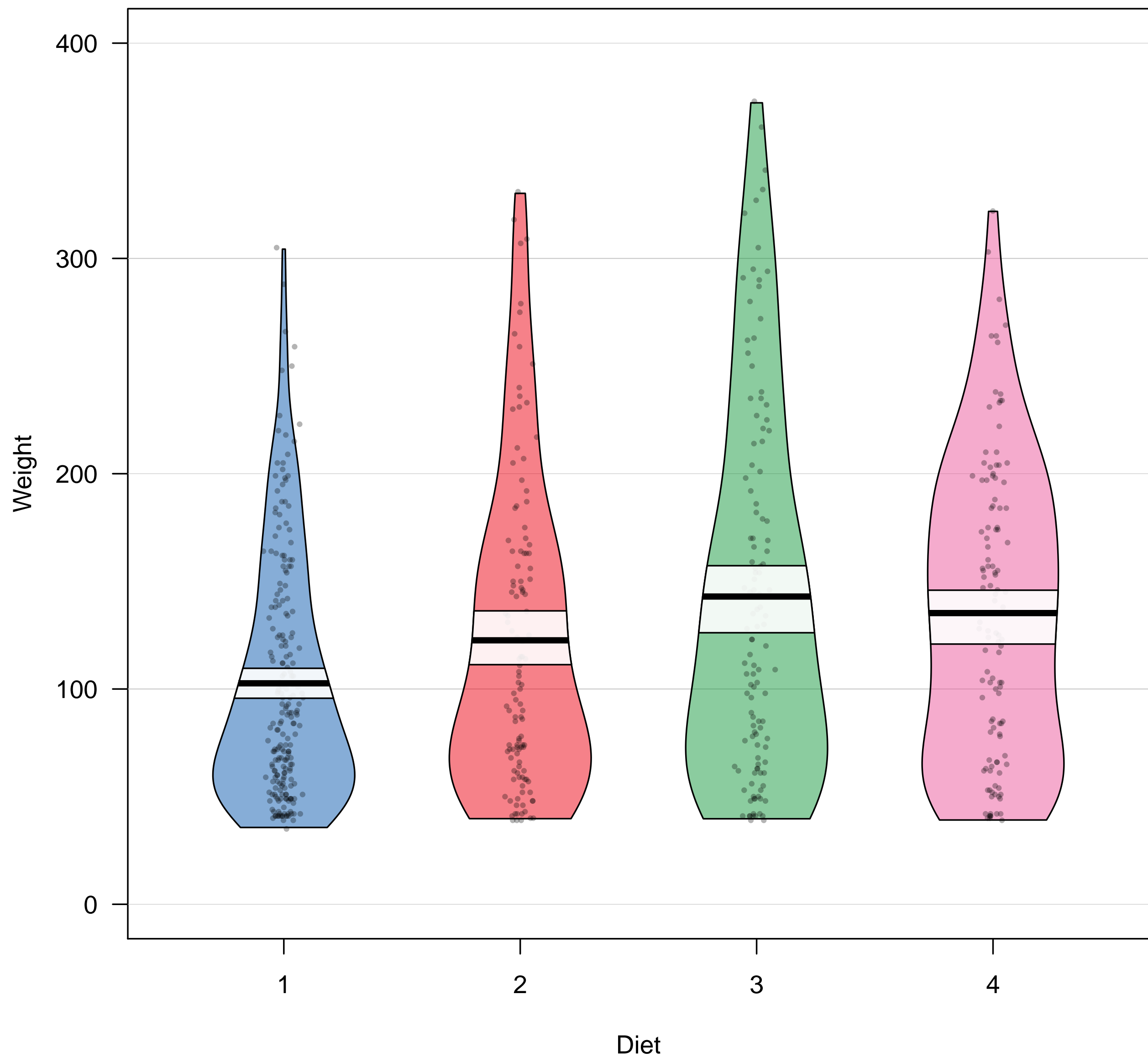
```
yarrr::pirateplot(formula = weight ~ Time, # dv is weight, iv is Diet
                  data = ChickWeight,
                  main = "Pirateplot of chicken weights",
                  xlab = "Diet",
                  ylab = "Weight")
```

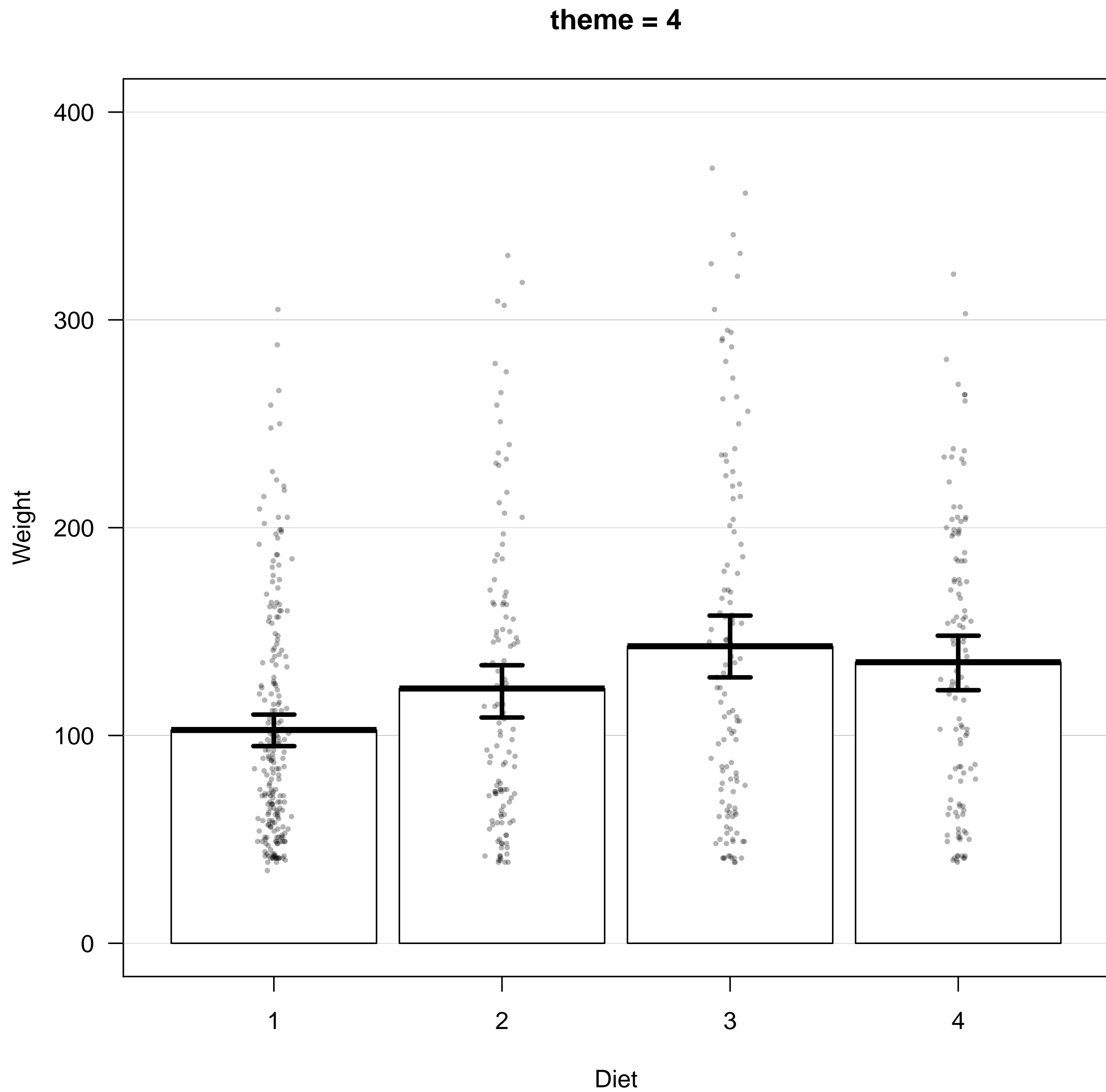
11.6.1 Pirateplot themes

There are many different pirateplot themes, these themes dictate the overall look of the plot. To specify a theme, just use the `theme = x` argument, where `x` is the theme number:




theme = 2
Weight
400
300
200
100
0
1
2
3
4
Diet


theme = 3
Weight
400
300
200
100
0
1
2
3
4
Diet

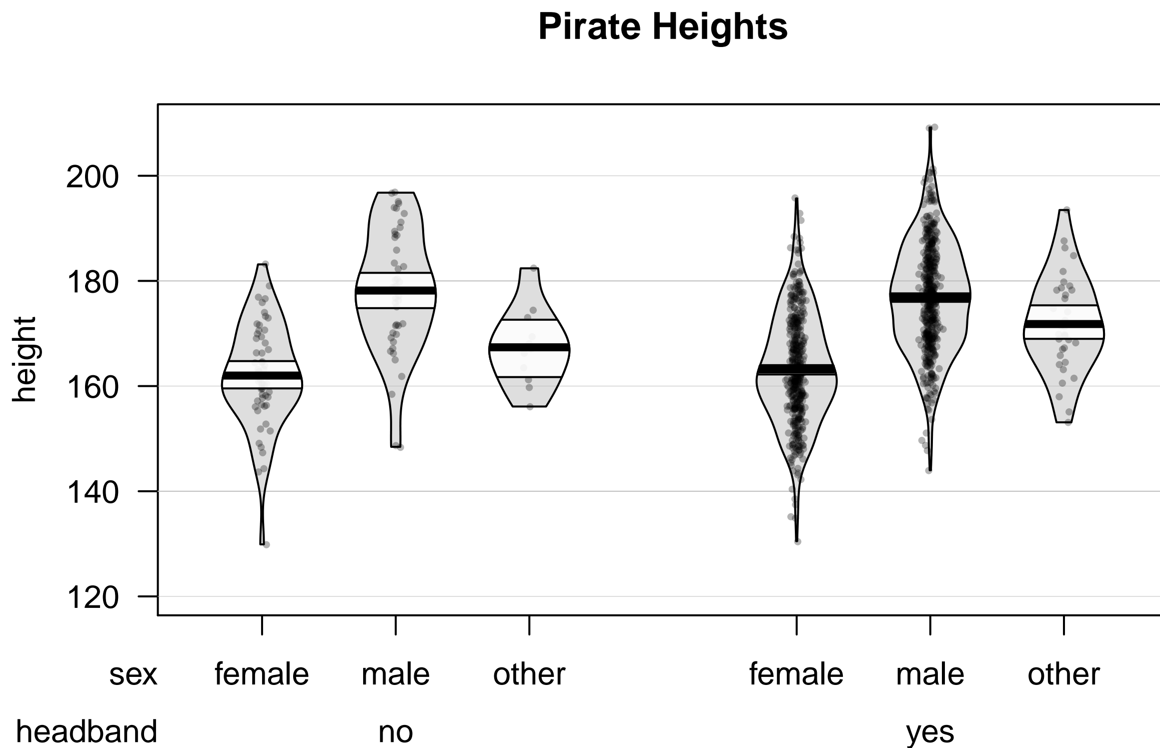


For example, here is a pirateplot height data from the `pirates` dataframe using `theme = 3`. Here, I'll plot pirates' heights as a function of their sex and whether or not they wear a headband. I'll also make the plot all grayscale by using the `pal = "gray"` argument:

```
yarrr::pirateplot(formula = height ~ sex + headband,    # DV = height, IV1 = sex, IV2 = headband
                  data = pirates,
                  theme = 3,
                  main = "Pirate Heights",
                  pal = "gray")
```

Table 11.6: Customising plotting elements

element	color	opacity
points	point.col, point.bg	point.o
beans	bean.f.col, bean.b.col	bean.f.o, bean.b.o
bar	bar.f.col, bar.b.col	bar.f.o, bar.b.o
inf	inf.f.col, inf.b.col	inf.f.o, inf.b.o
avg.line	avg.line.col	avg.line.o



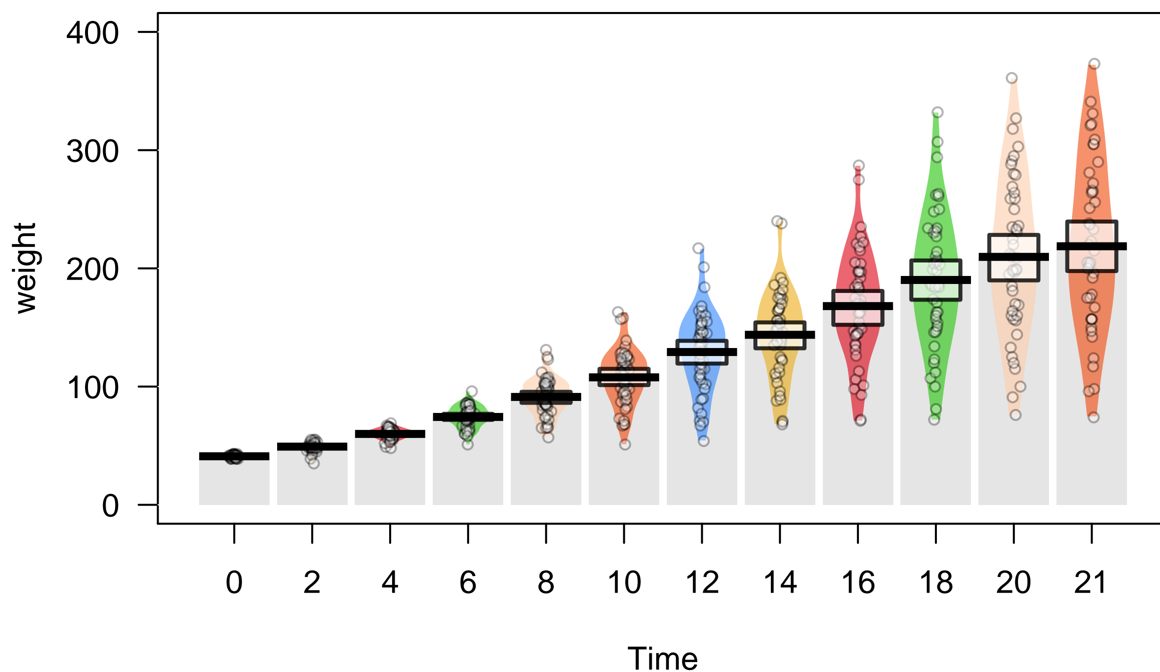
11.6.2 Customizing pirateplots

Regardless of the theme you use, you can always customize the color and opacity of graphical elements. To do this, specify one of the following arguments. Note: Arguments with `.f.` correspond to the *filling* of an element, while `.b.` correspond to the *border* of an element:

For example, I could create the following pirateplots using `theme = 0` and specifying elements explicitly:

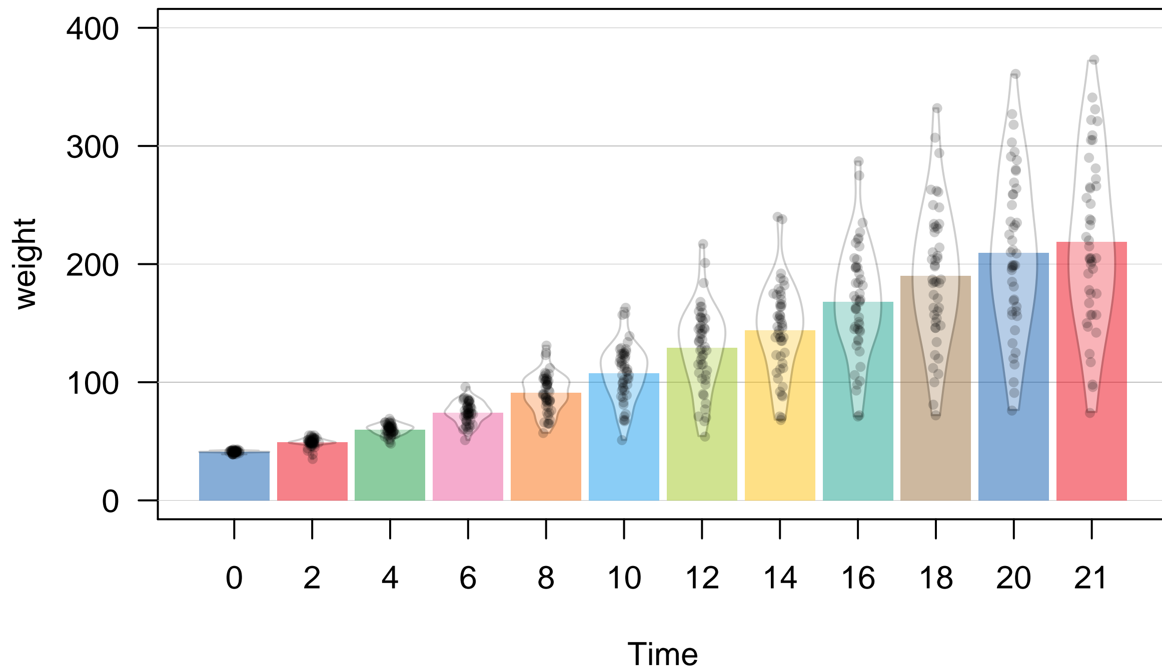
```
pirateplot(formula = weight ~ Time,
           data = ChickWeight,
           theme = 0,
           main = "Fully customized pirateplot",
           pal = "southpark", # southpark color palette
           bean.f.o = .6, # Bean fill
           point.o = .3, # Points
           inf.f.o = .7, # Inference fill
           inf.b.o = .8, # Inference border
           avg.line.o = 1, # Average line
           bar.f.o = .5, # Bar
```

```
          inf.f.col = "white", # Inf fill col
          inf.b.col = "black", # Inf border col
          avg.line.col = "black", # avg line col
          bar.f.col = gray(.8), # bar filling color
          point.pch = 21,
          point.bg = "white",
          point.col = "black",
          point.cex = .7)
```



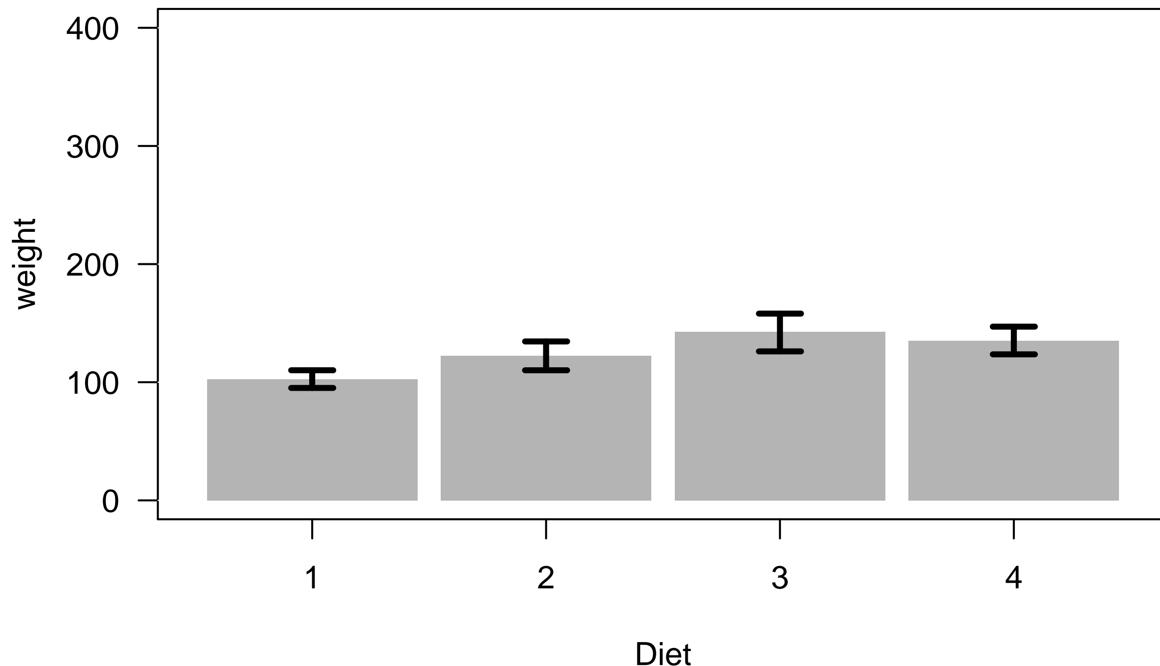
If you don't want to start from scratch, you can also start with a theme, and then make selective adjustments:

```
pirateplot(formula = weight ~ Time,
          data = ChickWeight,
          main = "Adjusting an existing theme",
          theme = 2,  # Start with theme 2
          inf.f.o = 0, # Turn off inf fill
          inf.b.o = 0, # Turn off inf border
          point.o = .2,   # Turn up points
          bar.f.o = .5, # Turn up bars
          bean.f.o = .4, # Light bean filling
          bean.b.o = .2, # Light bean border
          avg.line.o = 0, # Turn off average line
          point.col = "black") # Black points
```



Just to drive the point home, as a barplot is a special case of a pirateplot, you can even reduce a pirateplot into a horrible barplot:

```
# Reducing a pirateplot to a (at least colorful) barplot
pirateplot(formula = weight ~ Diet,
           data = ChickWeight,
           main = "Reducing a pirateplot to a (horrible) barplot",
           theme = 0,                                   # Start from scratch
           pal = "black",
           inf.disp = "line",                           # Use a line for inference
           inf.f.o = 1,                                 # Turn up inference opacity
           inf.f.col = "black",                         # Set inference line color
           bar.f.o = .3)
```



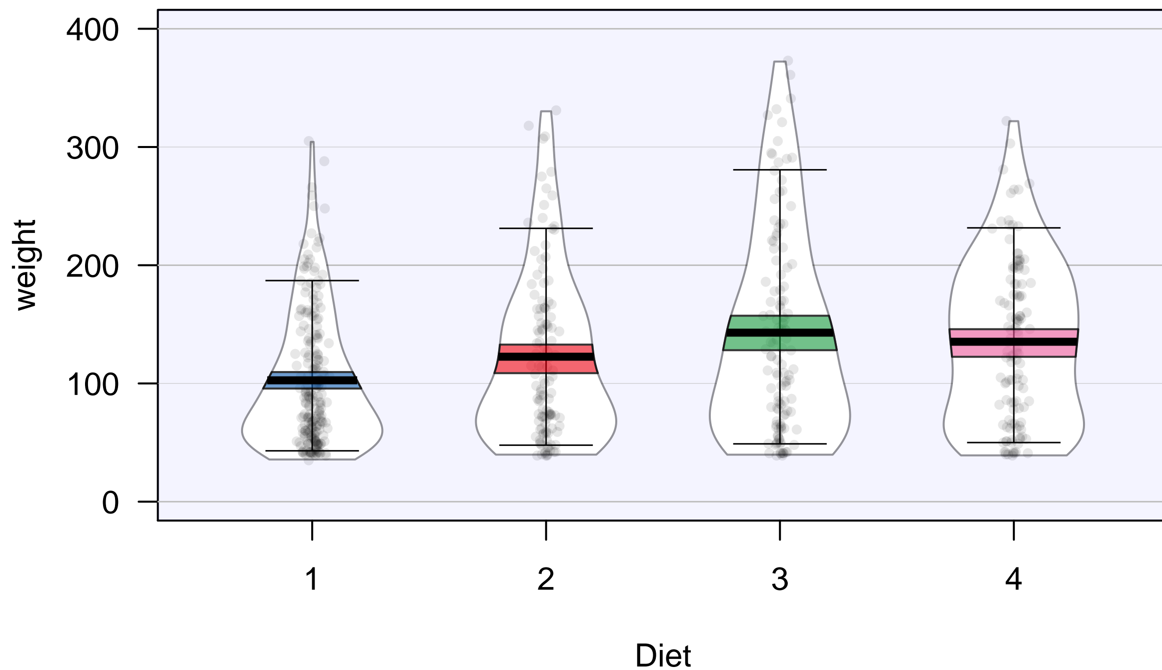
There are many additional arguments to `pirateplot()` that you can use to complete customize the look of your plot. To see them all, look at the help menu with `?pirateplot` or look at the vignette at

Table 11.7: Additional `pirateplot()` customizations.

Element	Argument	Examples
Background color	back.col	`back.col = 'gray(.9, .9)'`
Gridlines	gl.col, gl.lwd, gl.lty	`gl.col = 'gray', gl.lwd = c(.75, 0), gl.lty = 1`
Quantiles	quant, quant.lwd, quant.col	`quant = c(.1, .9), quant.lwd = 1, quant.col = 'black'`
Average line	avg.line.fun	`avg.line.fun = median`
Inference Calculation	inf.method	`inf.method = 'hdi'`, `inf.method = 'ci'`
Inference Display	inf.disp	`inf.disp = 'line'`, `inf.disp = 'bean'`, `inf.disp = 'rect'`

```
# Additional pirateplot customizations
pirateplot(formula = weight ~ Diet,
           data = ChickWeight,
           main = "Adding quantile lines and background colors",
           theme = 2,
           cap.beans = TRUE,
           back.col = transparent("blue", .95), # Add light blue background
           gl.col = "gray", # Gray gridlines
           gl.lwd = c(.75, 0),
           inf.f.o = .6, # Turn up inf filling
           inf.disp = "bean", # Wrap inference around bean
```

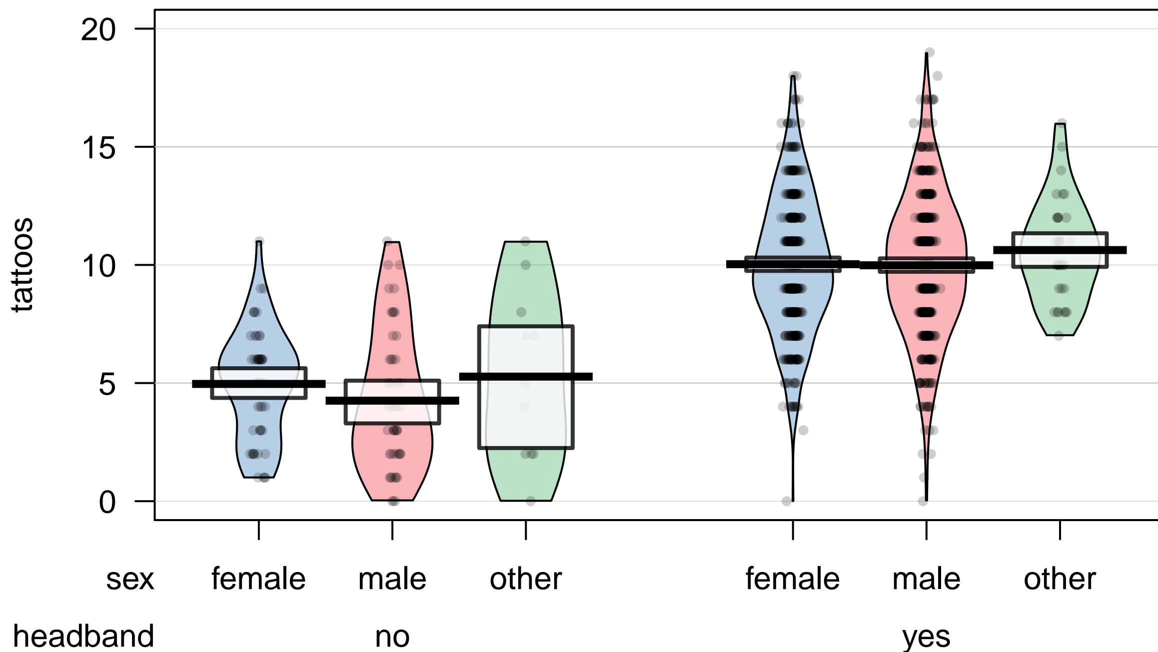
```
           bean.b.o = .4, # Turn down bean borders
           quant = c(.1, .9), # 10th and 90th quantiles
           quant.col = "black") # Black quantile lines
```



11.6.3 Saving output

If you include the `plot = FALSE` argument to a pirateplot, the function will return some values associated with each bean in the plot. In the next chunk, I'll

```
# Create a pirateplot
pirateplot(formula = tattoos ~ sex + headband,
           data = pirates)
```




```
# Save data from the pirateplot to an object
tattoos.pp <- pirateplot(formula = tattoos ~ sex + headband,
                         data = pirates,
                         plot = FALSE)
```

Now I can access the summary and inferential statistics from the plot in the `tattoos.pp` object. The most interesting element is `$summary` which shows summary statistics for each bean (aka, group):

```
# Show me statistics from groups in the pirateplot
tattoos.pp
## $summary
##      sex headband bean.num   n  avg inf.lb inf.ub
## 1 female       no        1  55  5.0    4.3    5.5
## 2   male       no        2  47  4.3    3.2    5.0
## 3  other       no        3  11  5.3    2.5    7.2
## 4 female      yes        4 409 10.0    9.8   10.3
## 5   male      yes        5 443 10.0    9.7   10.3
## 6  other      yes        6  35 10.6    9.9   11.4
##
## $avg.line.fun
## [1] "mean"
##
## $inf.method
## [1] "hdi"
##
## $inf.p
## [1] 0.95
```

Once you've created a plot with a high-level plotting function, you can add additional elements with *low-level* functions. For example, you can add data points with `points()`, reference lines with `abline()`, text with `text()`, and legends with `legend()`.

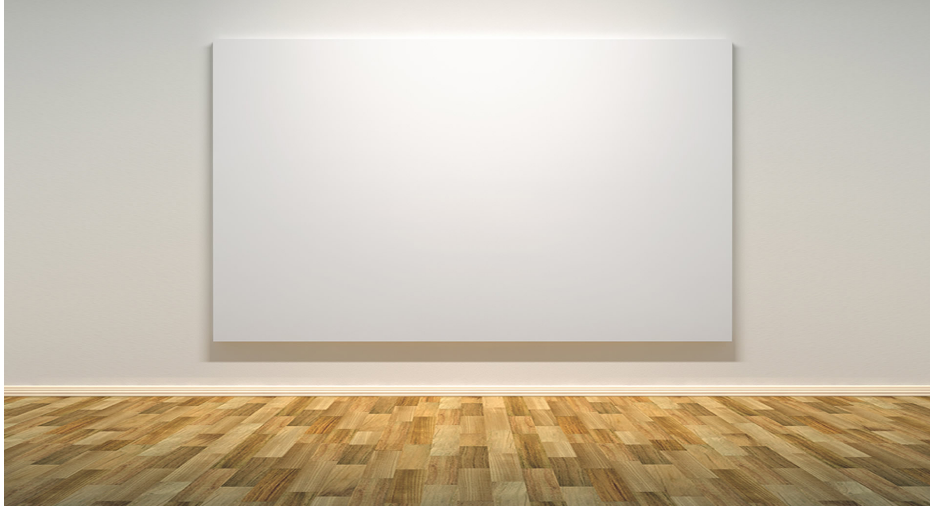


Figure 11.6: Sometimes it's nice to start with a blank plotting canvas, and then add each element individually with low-level plotting commands

11.7 Low-level plotting functions

Low-level plotting functions allow you to add elements, like points, or lines, to an existing plot. Here are the most common low-level plotting functions:

Table 11.8: Common low-level plotting functions.

Function	Outcome
`points(x, y)`	Adds points
`abline()`, `segments()`	Adds lines or segments
`arrows()`	Adds arrows
`curve()`	Adds a curve representing a function
`rect()`,`polygon()`	Adds a rectangle or arbitrary shape
`text()`, `mtext()`	Adds text within the plot, or to plot margins
`legend()`	Adds a legend
`axis()`	Adds an axis

11.7.1 Starting with a blank plot

Before you start adding elements with low-level plotting functions, it's useful to start with a blank plotting space like the one I have in Figure 11.7. To do this, execute the `plot()` function, but use the `type = "n"` argument to tell R that you don't want to plot anything yet. Once you've created a blank plot, you can additional elements with low-level plotting commands.

```
# Create a blank plotting space
plot(x = 1,
     xlab = "X Label",
     ylab = "Y Label",
     xlim = c(0, 100),
     ylim = c(0, 100),
     main = "Blank Plotting Canvas",
     type = "n")
```

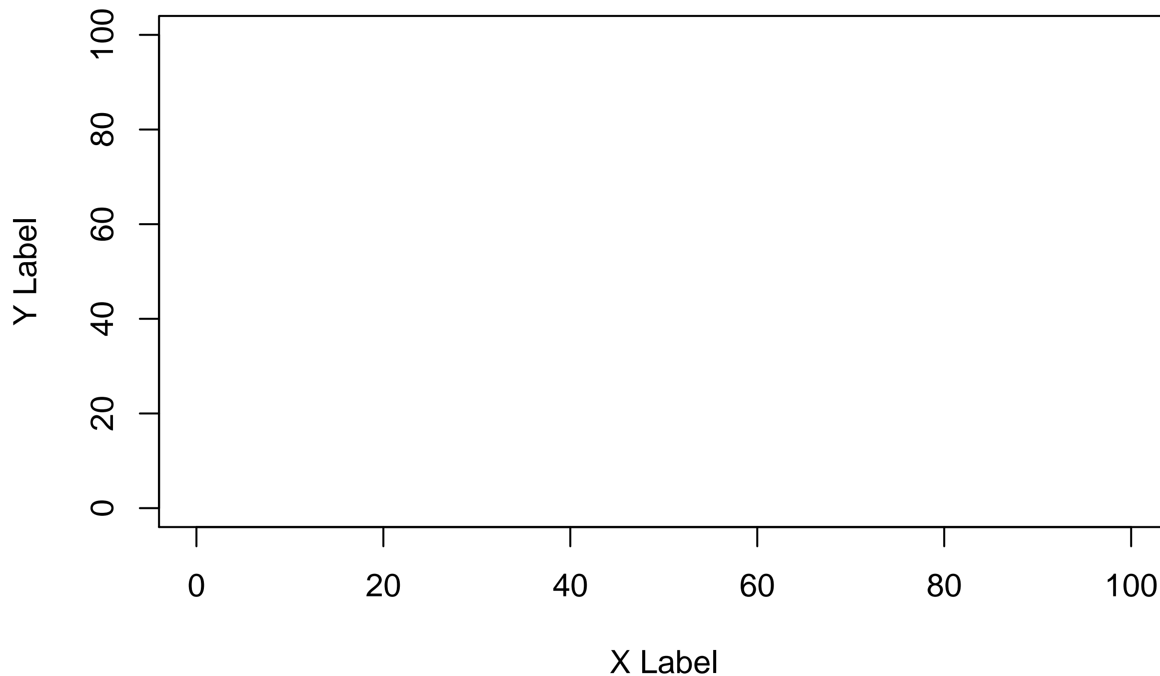


Figure 11.7: A blank plotting space, ready for additional elements!

11.7.2 points()

To add new points to an existing plot, use the `points()` function. The `points` function has many similar arguments to the `plot()` function, like `x` (for the x-coordinates), `y` (for the y-coordinates), and parameters like `col` (border color), `cex` (point size), and `pch` (symbol type). To see all of them, look at the help menu with `?points()`.

Let's use `points()` to create a plot with different symbol types for different data. I'll use the pirates dataset and plot the relationship between a pirate's age and the number of tattoos he/she has. I'll create separate points for male and female pirates:

```
# Create a blank plot
plot(x = 1,
     type = "n",
     xlim = c(100, 225),
     ylim = c(30, 110),
     pch = 16,
     xlab = "Height",
     ylab = "Weight",
     main = "Adding points to a plot with points()")

# Add coral2 points for male data
points(x = pirates$height[pirates$sex == "male"],
       y = pirates$weight[pirates$sex == "male"],
       pch = 16,
```

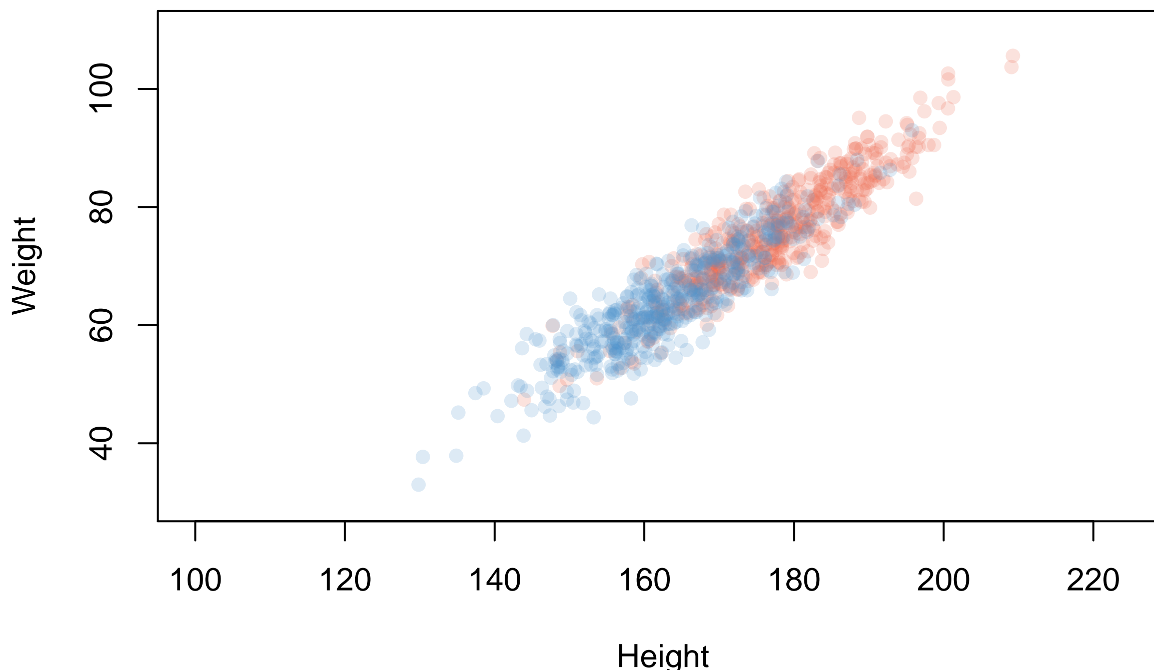


Figure 11.8: Using points() to add points with different colors

```
       col = transparent("coral2", trans.val = .8))

# Add steelblue points for female data
points(x = pirates$height[pirates$sex == "female"],
       y = pirates$weight[pirates$sex == "female"],
       pch = 16,
       col = transparent("steelblue3", trans.val = .8))
```

11.7.3 `abline()`, `segments()`, `grid()`

Table 11.9: Arguments to `abline()` and `segments()`

Argument	Outcome
`h, v`	Locations of horizontal and vertical lines (for `abline()` only)
`x0, y0, x1, y1`	Starting and ending coordinates of lines (for `segments()` only)
`lty`	Line type. 1 = solid, 2 = dashed, 3 = dotted, ...
`lwd`	Width of the lines specified by a number. 1 is the default (.2 is very thin, 5 is very thick)
`col`	Line color

To add straight lines to a plot, use `abline()` or `segments()`. `abline()` will add a line across the entire plot, while `segments()` will add a line with defined starting and end points.

For example, we can add reference lines to a plot with `abline()`. In the following plot, I'll add vertical and horizontal reference lines showing the means of the variables on the x and y axes, for the horizontal line, I'll specify `h = mean(pirates$height)`, for the vertical line, I'll specify `v = mean(pirates$weight)`

```
plot(x = pirates$weight,
     y = pirates$height,
```

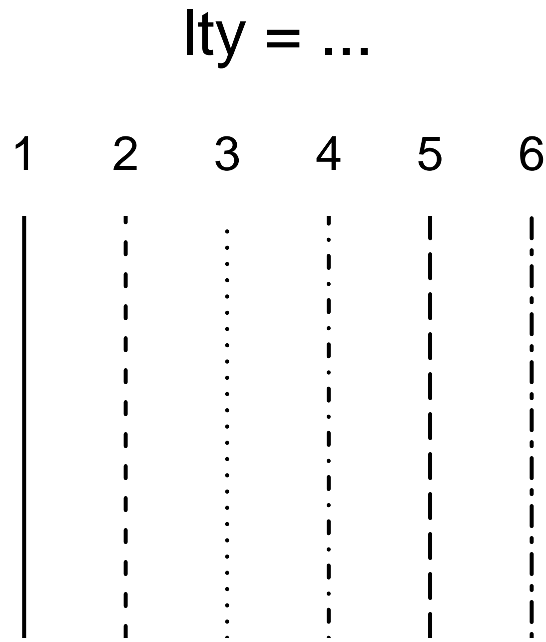
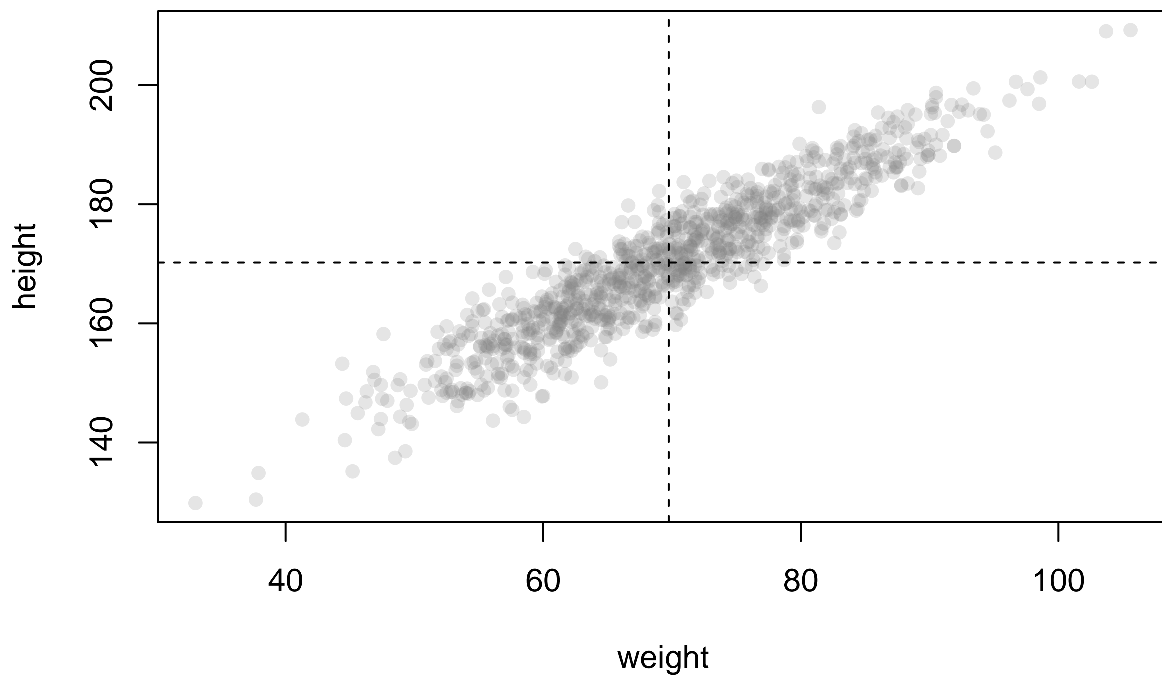


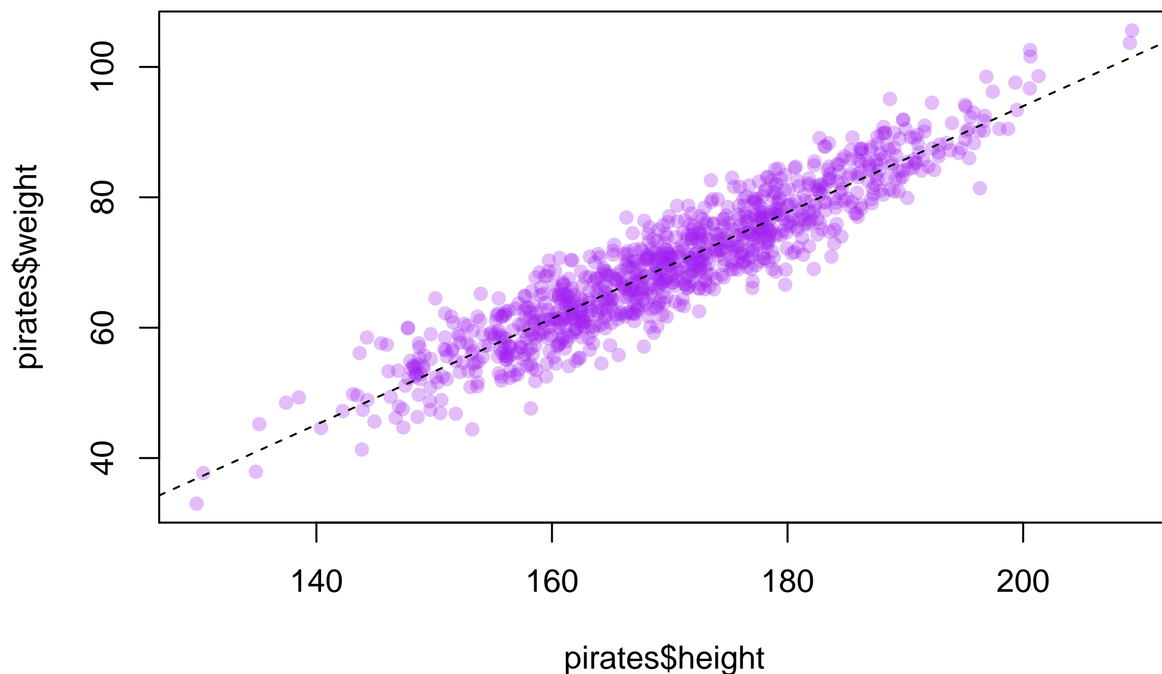
Figure 11.9: Changing line type with the lty argument.



To change the look of your lines, use the `lty` argument, which changes the type of line (see Figure 11.9), `lwd`, which changes its thickness, and `col` which changes its color

You can also add a regression line (also called a line of best fit) to a scatterplot by entering a regression object created with `lm()` as the main argument to `abline()`:

```
# Add a regression line to a scatterplot
plot(x = pirates$height,
     y = pirates$weight,
     pch = 16,
     col = transparent("purple", .7),
     main = "Adding a regression line to a scatterplot()")

# Add the regression line
abline(lm(weight ~ height, data = pirates),
       lty = 2)
```



The `segments()` function works very similarly to `abline()` – however, with the `segments()` function, you specify the beginning and end points of the segments with the arguments `x0`, `y0`, `x1`, and `y1`. In Figure 11.10 I use `segments()` to connect two vectors of data:

```
# Before and after data
before <- c(2.1, 3.5, 1.8, 4.2, 2.4, 3.9, 2.1, 4.4)
after <- c(7.5, 5.1, 6.9, 3.6, 7.5, 5.2, 6.1, 7.3)

# Create plotting space and before scores
plot(x = rep(1, length(before)),
     y = before,
     xlim = c(.5, 2.5),
     ylim = c(0, 11),
     ylab = "Score",
     xlab = "Time",
     main = "Using segments() to connect points",
     xaxt = "n")

# Add after scores
```

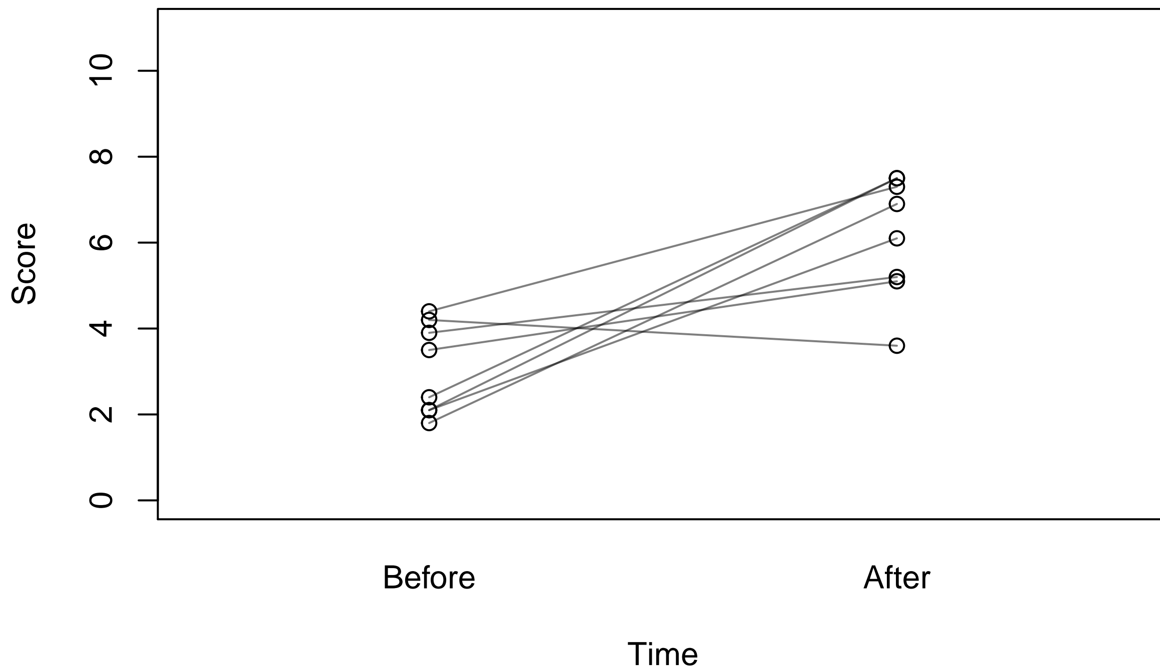
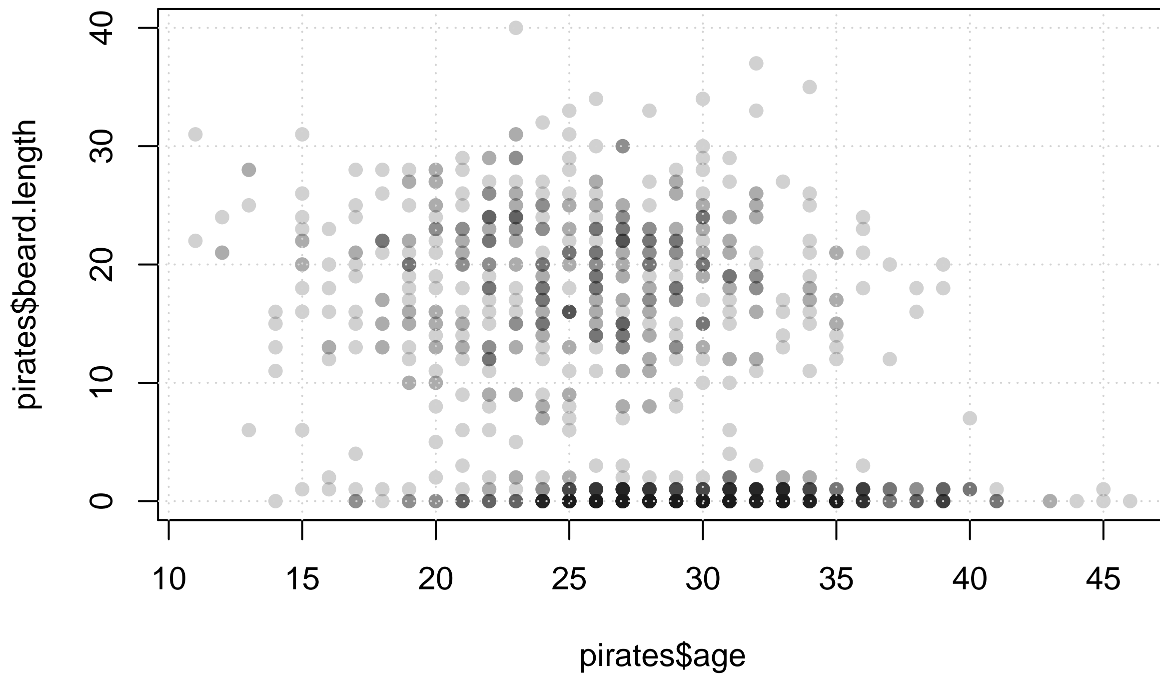


Figure 11.10: Connecting points with segments().

```
points(x = rep(2, length(after)), y = after)

# Add connections with segments()
segments(x0 = rep(1, length(before)),
         y0 = before,
         x1 = rep(2, length(after)),
         y1 = after,
         col = gray(0, .5))

# Add labels
mtext(text = c("Before", "After"),
      side = 1, at = c(1, 2), line = 1)
```

The `grid()` function allows you to easily add grid lines to a plot (you can customize your grid lines further with `lty`, `lwd`, and `col` arguments):

```
# Add gridlines to a plot with grid()
plot(pirates$age,
     pirates$beard.length,
     pch = 16,
     col = gray(.1, .2), main = "Add grid lines to a plot with grid()")

# Add gridlines
grid()
```



11.7.4 `text()`

Table 11.10: Arguments to `text()`

Argument	Outcome
`x, y`	Coordinates of the labels
`labels`	Labels to be plotted
`cex`	Size of the labels
`adj`	Horizontal text adjustment. `adj = 0` is left justified, `adj = .5` is centered, and `adj = 1` is right-justified
`pos`	Position of the labels relative to the coordinates. `pos = 1`, puts the label below the coordinates, while 2, 3, and 4 put it to the left, top and right of the coordinates respectively

With `text()`, you can add text to a plot. You can use `text()` to highlight specific points of interest in the plot, or to add information (like a third variable) for every point in a plot. For example, the following code adds the three words "Put", "Text", and "Here" at the coordinates (1, 9), (5, 5), and (9, 1) respectively. See Figure 11.11 for the plot:

```
plot(1,
     xlim = c(0, 10),
     ylim = c(0, 10),
     type = "n")

text(x = c(1, 5, 9),
     y = c(9, 5, 1),
     labels = c("Put", "text", "here"))
```

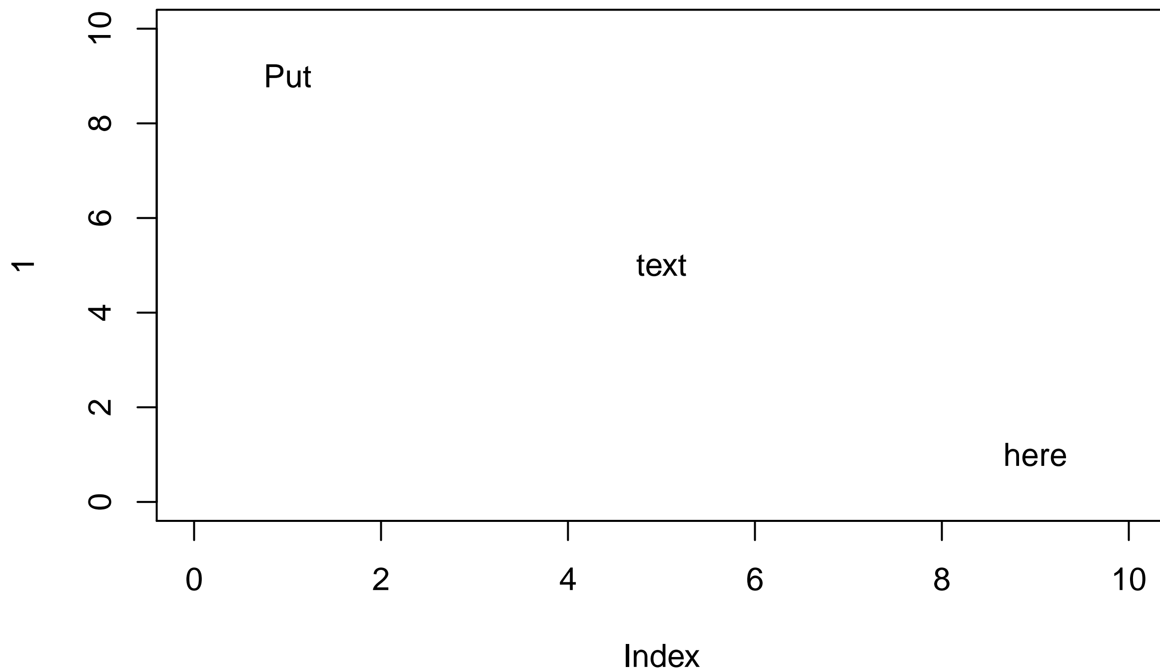


Figure 11.11: Adding text to a plot with text()

You can do some cool things with `text()`, in Figure 11.12 I create a scatterplot of data, and add data labels above each point by including the `pos = 3` argument:

```
# Create data vectors
height <- c(156, 175, 160, 172, 159, 165, 178)
weight <- c(65, 74, 69, 72, 66, 75, 75)
id <- c("andrew", "heidi", "becki", "madisen", "david", "vincent", "jack")

# Plot data
plot(x = height,
     y = weight,
     xlim = c(155, 180),
     ylim = c(65, 80),
     pch = 16,
     col = yarrr::piratepal("xmen"))

# Add id labels
text(x = height,
     y = weight,
     labels = id,
     pos = 3)            # Put labels above the points
```

When entering text in the `labels` argument, keep in mind that R will, by default, plot the entire text in one line. However, if you are adding a long text string (like a sentence), you may want to separate the text into separate lines. To do this, add the text `\n` where you want new lines to start. Look at Figure 11.13 for an example.

```
plot(1,
     type = "n",
     main = "The \\n tag",
     xlab = "", ylab = "")
```

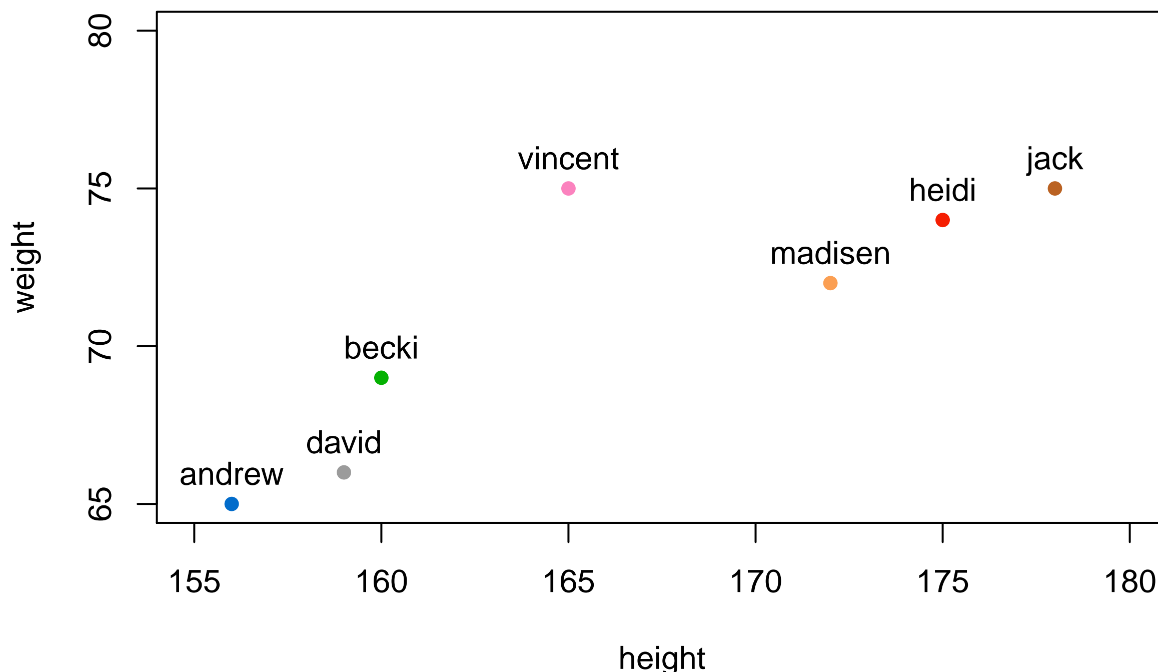


Figure 11.12: Adding labels to points with text()

```
# Text withoutbreaks
text(x = 1, y = 1.3, labels = "Text without \\n", font = 2)
text(x = 1, y = 1.2,
     labels = "Haikus are easy. But sometimes they don't make sense. Refrigerator",
     font = 3) # italic font

abline(h = 1, lty = 2)
# Text with  breaks
text(x = 1, y = .92, labels = "Text with \\n", font = 2)
text(x = 1, y = .7,
     labels = "Haikus are easy\nBut sometimes they don't make sense\nRefrigerator",
     font = 3)   # italic font
```

11.7.5 Combining text and numbers with paste()

A common way to use text in a plot, either in the main title of a plot or using the `text()`function, is to combine text with numerical data. For example, you may want to include the text "Mean = 3.14" in a plot to show that the mean of the data is 3.14. But how can we combine numerical data with text? In R, we can do this with the `paste()` function:

The paste function will be helpful to you anytime you want to combine either multiple strings, or text and strings together. For example, let's say you want to write text in a plot that says `The mean of these data are XXX`, where XXX is replaced by the group mean. To do this, just include the main text and the object referring to the numerical mean as arguments to `paste()`. In Figure X I plot the chicken weights over time, and add text to the plot specifying the overall mean of weights.

```
# Create the plot
plot(x = ChickWeight$Time,
```

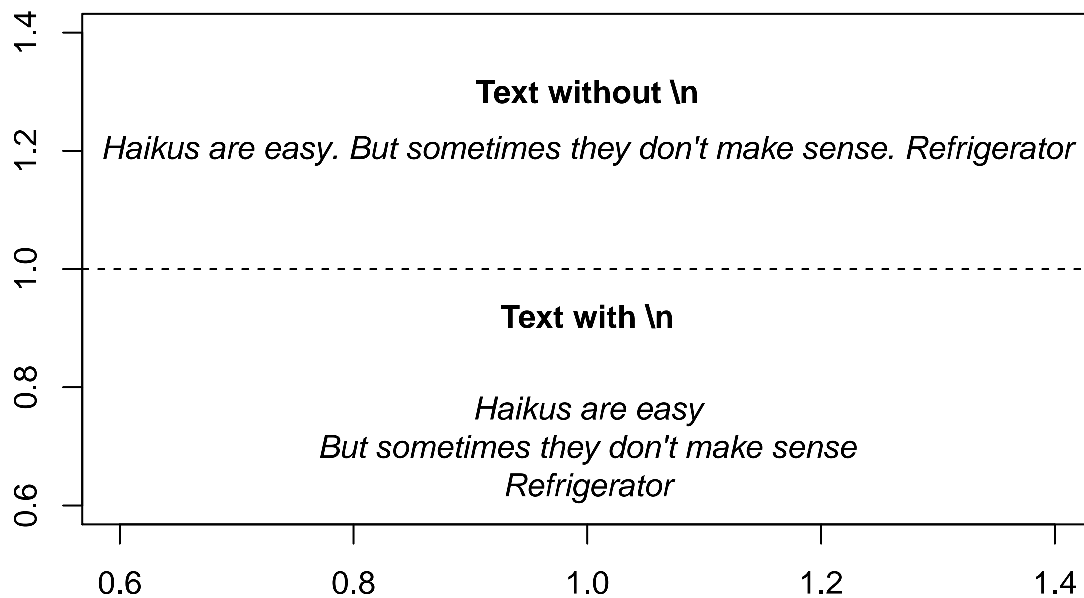
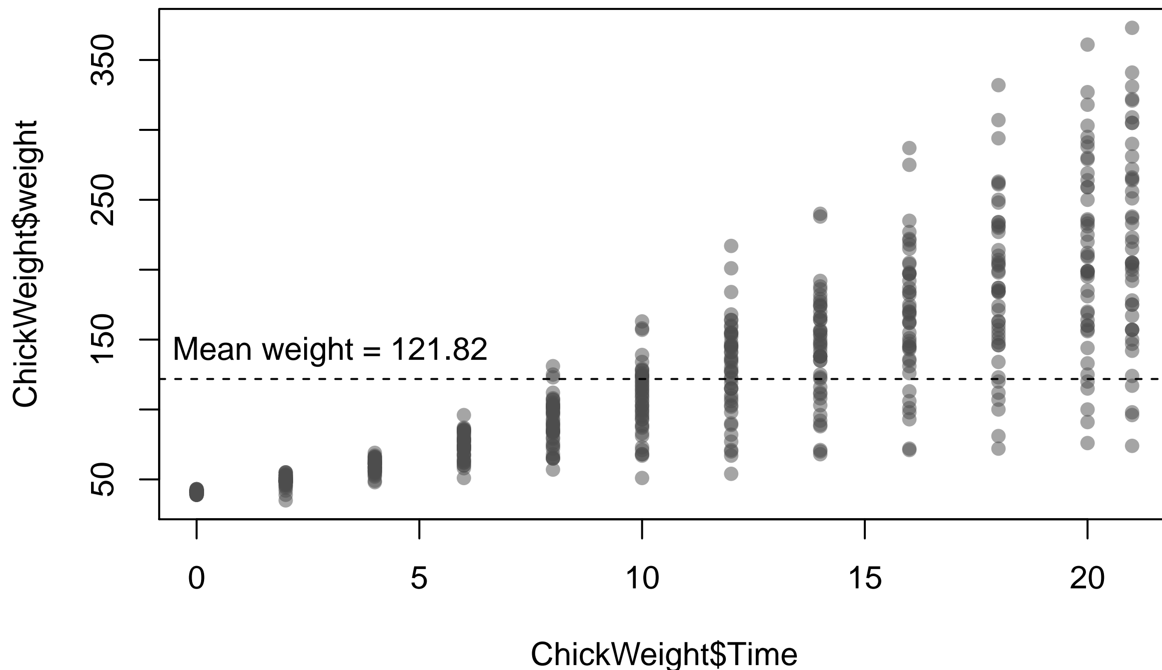


Figure 11.13: Break up lines in text with .

```
     y = ChickWeight$weight,
     col = gray(.3, .5),
     pch = 16,
     main = "Combining text with numeric scalers using paste()")

# Add reference line
abline(h = mean(ChickWeight$weight),
       lty = 2)

# Add text
text(x = 3,
     y = mean(ChickWeight$weight),
     labels = paste("Mean weight =",
                    round(mean(ChickWeight$weight), 2)),
     pos = 3)
```



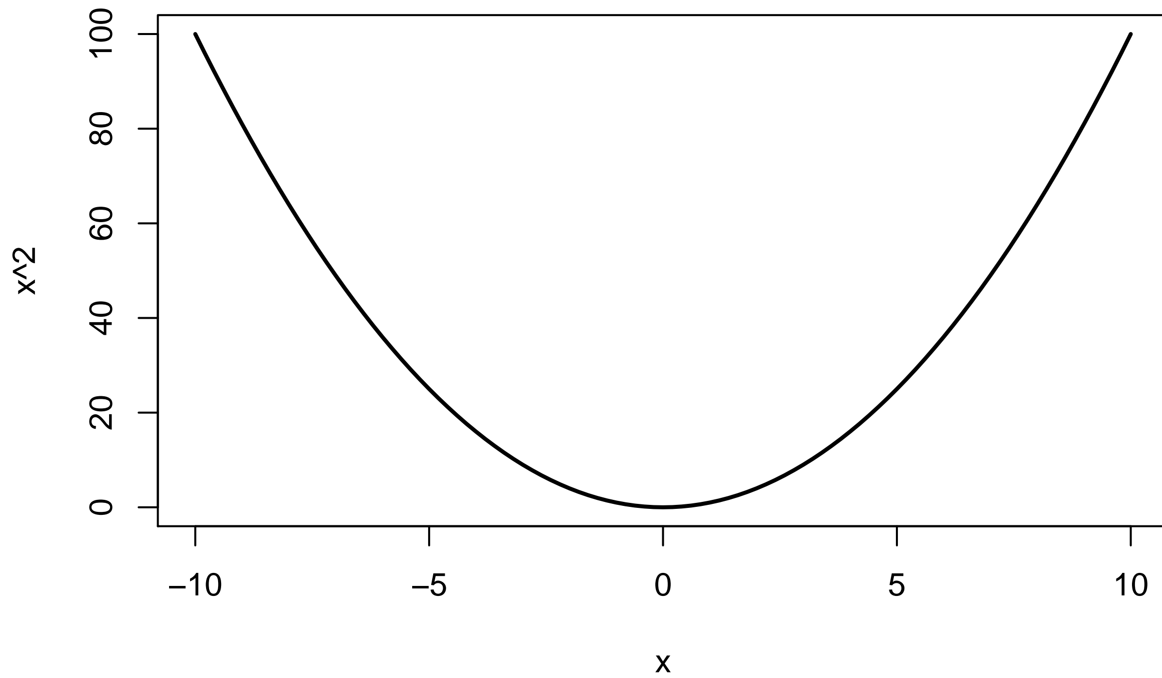
11.7.6 curve()

Table 11.11: Arguments to curve()

Argument	Outcome
expr	The name of a function written as a function of x that returns a single vector. You can either use base functions in R like expr = x^2, expr = x + 4 - 2, or use your own custom functions such as expr = my.fun, where my.fun is previously defined (e.g.; my.fun <- function(x) {dnorm(x, mean = 10, sd = 3))
from, to	The starting (from) and ending (to) value of x to be plotted.
add	A logical value indicating whether or not to add the curve to an existing plot. If add = FALSE, then curve() will act like a high-level plotting function and create a new plot. If add = TRUE, then curve() will act like a low-level plotting function.
lty, lwd, col	Additional standard line arguments

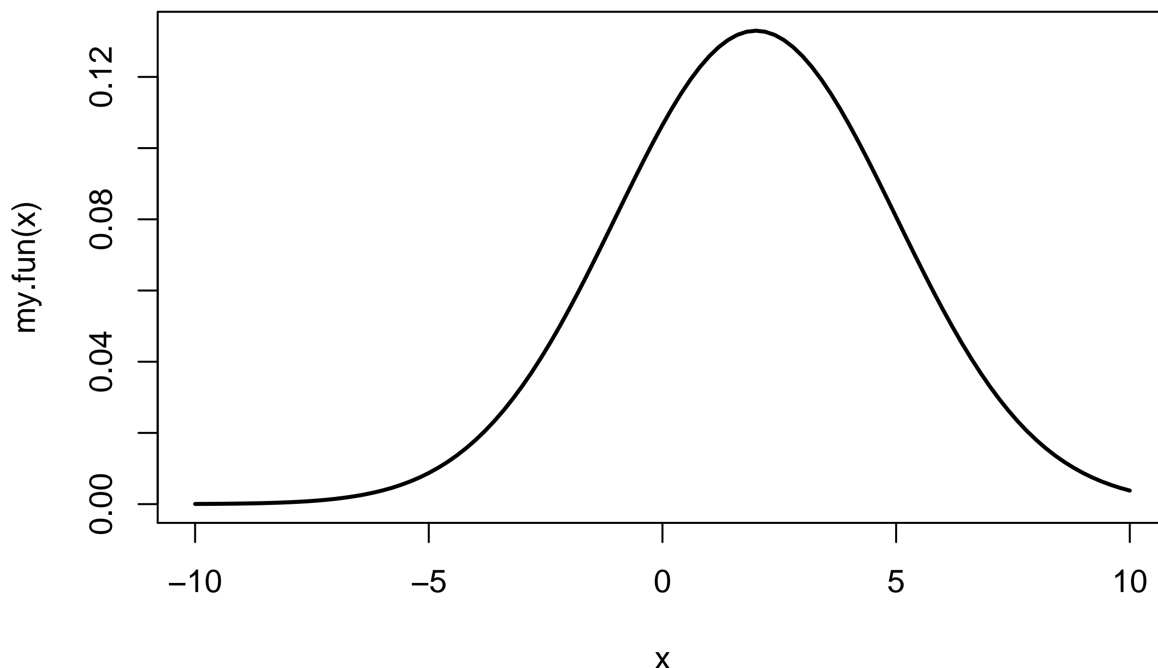
The curve() function allows you to add a line showing a specific function or equation to a plot. For example, to add the function x^2 to a plot from the x-values -10 to 10, you can run the code:

```
# Plot the function x^2 from -10 to +10
curve(expr = x^2,
      from = -10,
      to = 10, lwd = 2)
```



If you want to add a custom function to a plot, you can define the function and then use that function name as the argument to `expr`. For example, to plot the normal distribution with a mean of 10 and standard deviation of 3, you can use this code:

```
# Plot the normal distribution with mean = 22 and sd = 3

# Create a function
my.fun <- function(x) {dnorm(x, mean = 2, sd = 3)}

curve(expr = my.fun,
      from = -10,
      to = 10, lwd = 2)
```



In Figure~11.14, I use the `curve()` function to create curves of several mathematical formulas.

```
# Create plotting space
plot(1,
     xlim = c(-5, 5), ylim = c(-5, 5),
     type = "n",
     main = "Plotting function lines with curve()",
     ylab = "", xlab = "")

# Add x and y-axis lines
abline(h = 0)
abline(v = 0)

# set up colors
col.vec <- piratepal("google")

# x ^ 2
curve(expr = x^2, from = -5, to = 5,
      add = TRUE, lwd = 3, col = col.vec[1])

# sin(x)
curve(expr = sin, from = -5, to = 5,
      add = TRUE, lwd = 3, col = col.vec[2])

# dnorm(mean = 2, sd = .2)
my.fun <- function(x) {return(dnorm(x, mean = 2, sd = .2))}
curve(expr = my.fun,
      from = -5, to = 5,
      add = TRUE,
      lwd = 3, col = col.vec[3])

# Add legend
legend("bottomright",
```

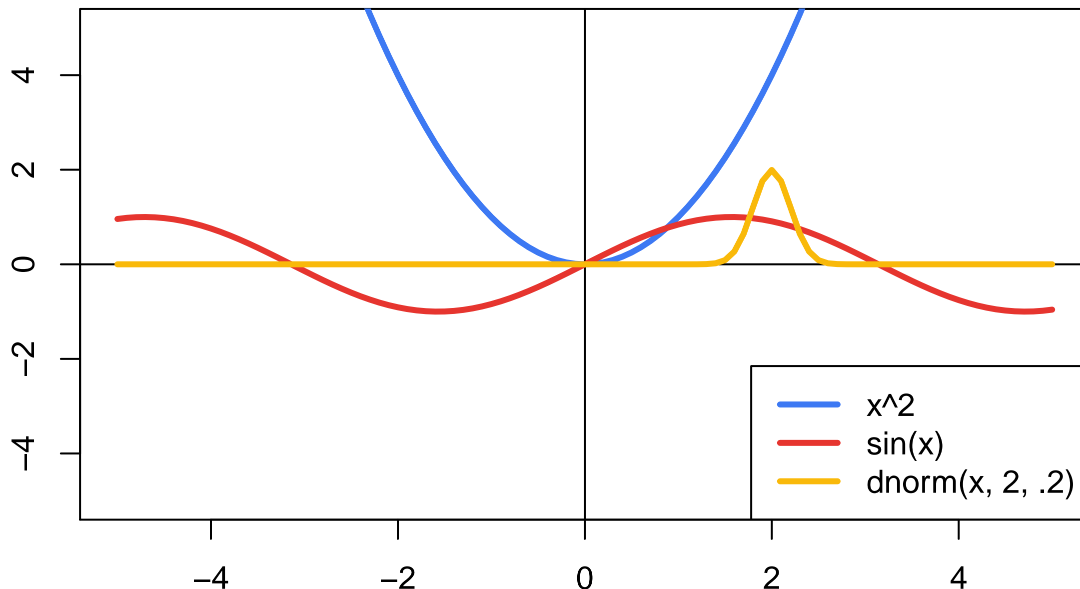


Figure 11.14: Drawing function lines with curve()

```
       legend = c("x^2", "sin(x)", "dnorm(x, 2, .2)"),
       col = col.vec[1:3],
       lwd = 3)
```

11.7.7 legend()

Table 11.12: Arguments to `legend()`

Argument	Outcome
`x, y`	Coordinates of the legend - for example, `x = 0, y = 0` will put the text at the coordinates (0, 0). Alternatively, you can enter a string indicating where to put the legend (i.e.; `"topright"`, `"topleft"`). For example, `"bottomright"` will always put the legend at the bottom right corner of the plot.
`labels`	A string vector specifying the text in the legend. For example, `legend = c("Males, "Females")` will create two groups with names Males and Females.
`pch, lty, lwd, col, pt.bg, ...`	Additional arguments specifying symbol types (`pch`), line types (`lty`), line widths (`lwd`), background color of symbol types 21 through 25 (`pt.bg`) and several other optional arguments. See `?legend` for a complete list

The last low-level plotting function that we'll go over in detail is `legend()` which adds a legend to a plot. For example, to add a legend to to bottom-right of an existing graph where data from females are plotted in blue circles and data from males are plotted in pink circles, you'd use the following code:

```
# Add a legend to the bottom right of a plot

legend("bottomright",                  # Put legend in bottom right of graph
       legend = c("Females", "Males"), # Names of groups
       col = c("blue", "orange"),      # Colors of symbols
       pch = c(16, 16))                # Symbol types
```

In Figure 11.15 I use this code to add a legend to plot containing data from males and females:

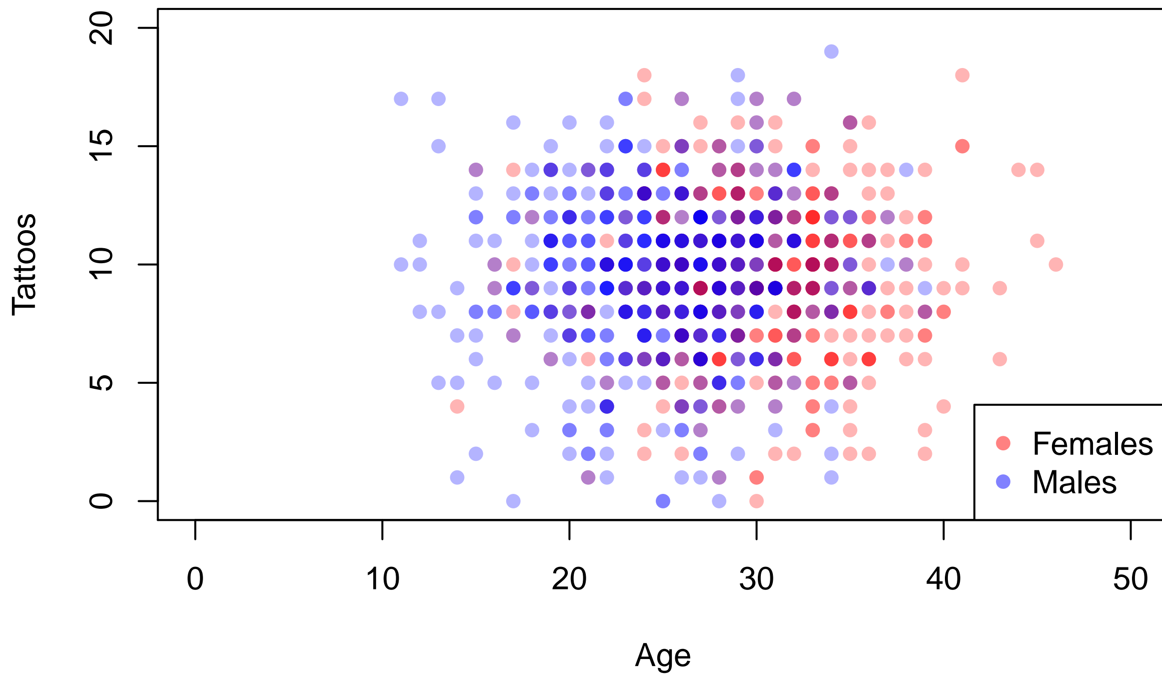


Figure 11.15: Adding a legend to a plot with legend().

```
       pch = c(16, 16),
       bg = "white")
```

There are many more low-level plotting functions that can add additional elements to your plots. Here are some I use. To see examples of how to use each one, check out their associated help menus.

```
plot(1, xlim = c(1, 100), ylim = c(1, 100),
     type = "n", xaxt = "n", yaxt = "n",
     ylab = "", xlab = "", main = "Adding simple figures to a plot")

text(25, 95, labels = "rect()")
rect(xleft = 10, ybottom = 70,
     xright = 40, ytop = 90, lwd = 2, col = "coral")

text(25, 60, labels = "polygon()")
polygon(x = runif(6, 15, 35),
        y = runif(6, 40, 55),
        col = "skyblue")

text(25, 30, labels = "segments()")
segments(x0 = runif(5, 10, 40),
         y0 = runif(5, 5, 25),
         x1 = runif(5, 10, 40),
         y1 = runif(5, 5, 25),
         lwd = 2)

text(75, 95, labels = "symbols(circles)")
```

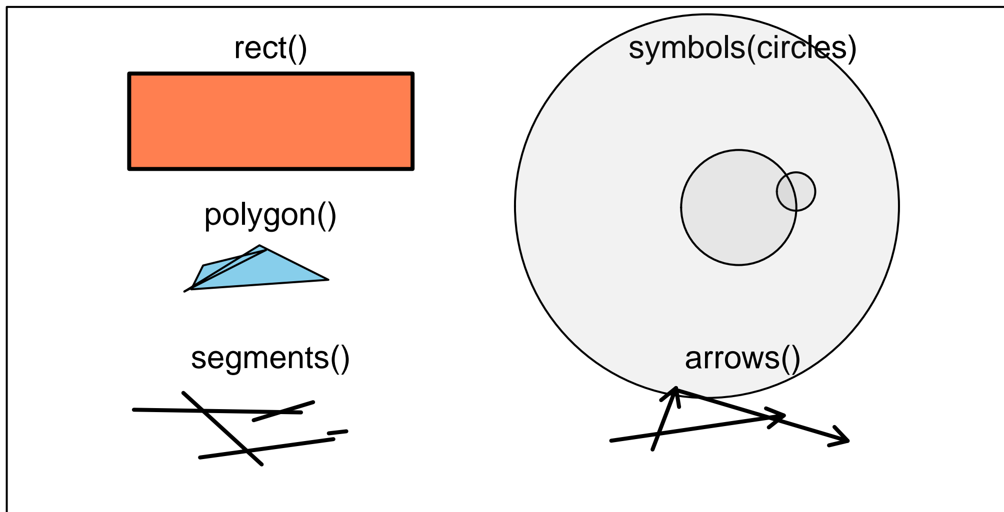


Figure 11.16: Additional figures one can add to a plot with rect(), polygon(), segments(), symbols(), and arrows().

```
symbols(x = runif(3, 60, 90),
        y = runif(3, 60, 70),
        circles = c(1, .1, .3),
        add = TRUE, bg = gray(.5, .1))

text(75, 30, labels = "arrows()")
arrows(x0 = runif(3, 60, 90),
       y0 = runif(3, 10, 25),
       x1 = runif(3, 60, 90),
       y1 = runif(3, 10, 25),
       length = .1, lwd = 2)
```

11.8 Saving plots to a file with `pdf()`, `jpeg()` and `png()`

Once you've created a plot in R, you may wish to save it to a file so you can use it in another document. To do this, you'll use either the `pdf()`, `png()` or `jpeg()` functions. These functions will save your plot to either a .pdf, .jpg, or .png file.

Table 11.13: Arguments to `pdf()`, `jpeg()` and `png()`

Argument	Outcome
`file`	The directory and name of the final plot entered as a string. For example, to put a plot on my desktop, I'd write `file = "/Users/nphillips/Desktop/plot.pdf"` when creating a pdf, and `file = "/Users/nphillips/Desktop/plot.jpg"` when creating a jpeg.
`width, height`	The width and height of the final plot in inches.

Argument	Outcome
`dev.off()`	This is *not* an argument to `pdf()` and `jpeg()`. You just need to execute this code after creating the plot to finish creating the image file (see examples).

To use these functions to save files, you need to follow 3 steps:

1. Execute the `pdf()` or `jpeg()` functions with `file, width, height` arguments.
2. Execute all your plotting code (e.g.; `plot(x = 1:10, y = 1:10)`)
3. Complete the file by executing the command `dev.off()`. This tells R that you're done creating the file.

The chunk below shows an example of the three steps in creating a pdf:

```
# Step 1: Call the pdf command to start the plot
pdf(file = "/Users/ndphillips/Desktop/My Plot.pdf",   # The directory you want to save the file in
    width = 4, # The width of the plot in inches
    height = 4) # The height of the plot in inches

# Step 2: Create the plot with R code
plot(x = 1:10,
     y = 1:10)
abline(v = 0) # Additional low-level plotting commands
text(x = 0, y = 1, labels = "Random text")

# Step 3: Run dev.off() to create the file!
dev.off()
```

You'll notice that after you close the plot with `dev.off()`, you'll see a message in the prompt like "null device". That's just R telling you that you can now create plots in the main R plotting window again.

The functions `pdf()`, `jpeg()`, and `png()` all work the same way, they just return different file types. If you can, use `pdf()` it saves the plot in a high quality format.

11.9 Examples

Figure 11.17 shows a modified version of a scatterplot I call a `balloonplot`:

```
# Turn a boring scatterplot into a  balloonplot!

# Create some random correlated data
x <- rnorm(50, mean = 50, sd = 10)
y <- x + rnorm(50, mean = 20, sd = 8)

# Set up the plotting space
plot(1,
     bty = "n",
     xlim = c(0, 100),
     ylim = c(0, 100),
     type = "n", xlab = "", ylab = "",
     main = "Turning a scatterplot into a balloon plot!")

# Add gridlines
grid()
```

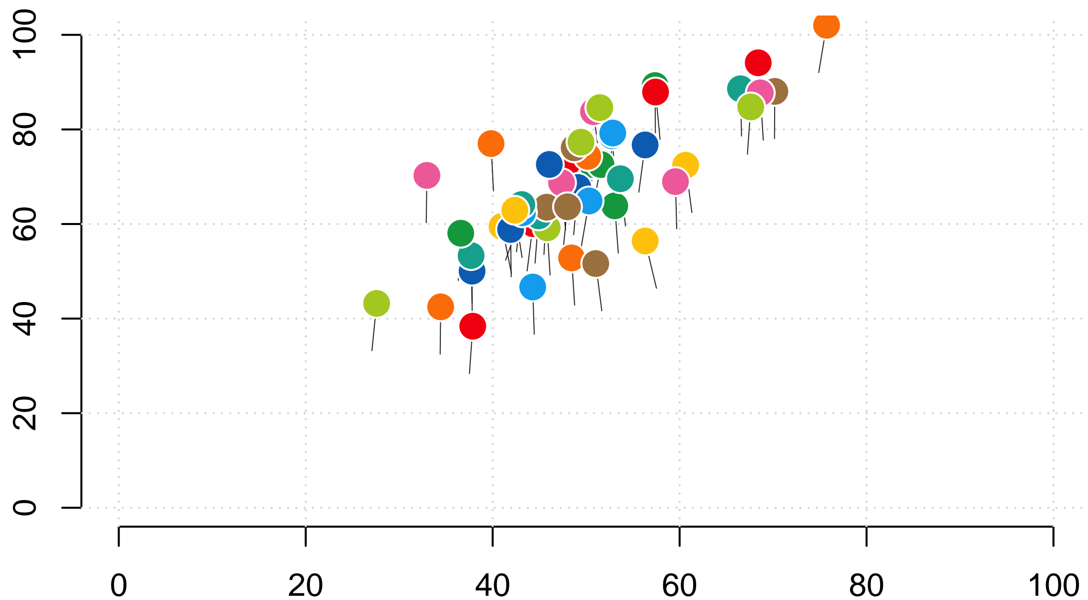


Figure 11.17: A balloon plot

```
# Add Strings with segments()
segments(x0 = x + rnorm(length(x), mean = 0, sd = .5),
         y0 = y - 10,
         x1 = x,
         y1 = y,
         col = gray(.1, .95),
         lwd = .5)

# Add balloons
points(x, y,
       cex = 2, # Size of the balloons
       pch = 21,
       col = "white", # white border
       bg = yarrr::piratepal("basel"))  # Filling color
```

You can use colors and point sizes in a scatterplot to represent third variables. In Figure 11.18, I'll plot the relationship between pirate height and weight, but now I'll make the size and color of each point reflect how many tattoos the pirate has

```
# Just the first 100 pirates
pirates.r <- pirates[1:100,]

plot(x = pirates.r$height,
     y = pirates.r$weight,
     xlab = "height",
     ylab = "weight",
     main = "Specifying point sizes and colors with a 3rd variable",
     cex = pirates.r$tattoos  / 8,          # Point size reflects how many tattoos they have
     col = gray(1 - pirates.r$tattoos / 20)) # color reflects tattoos
```

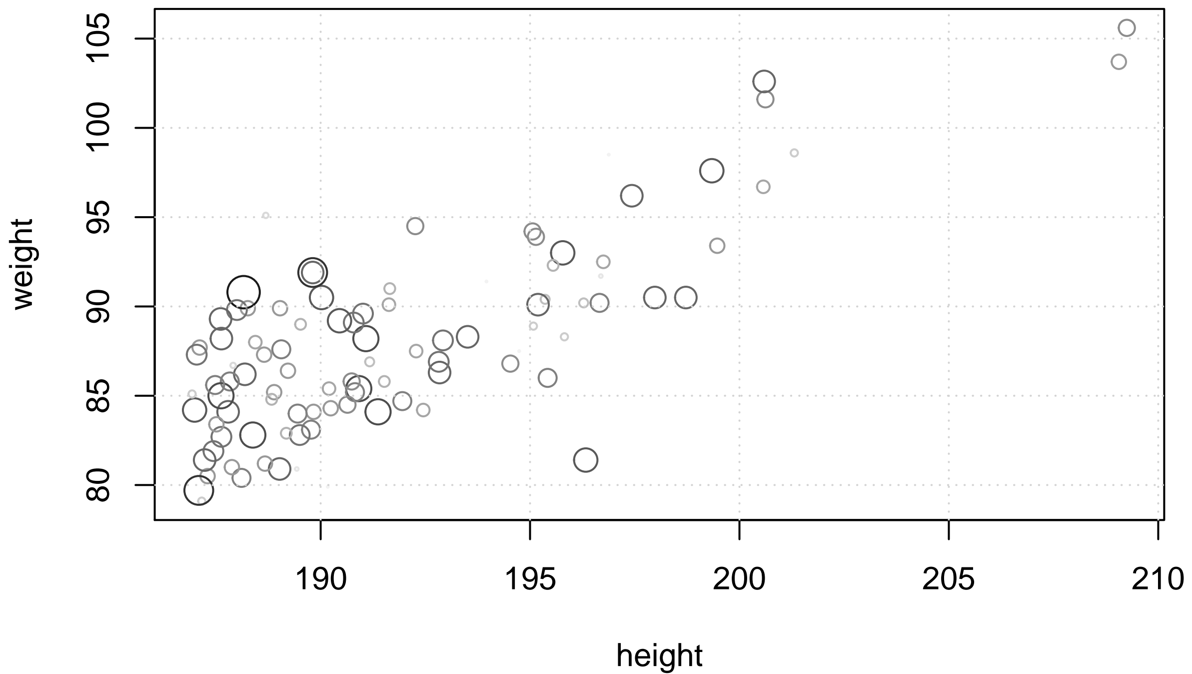
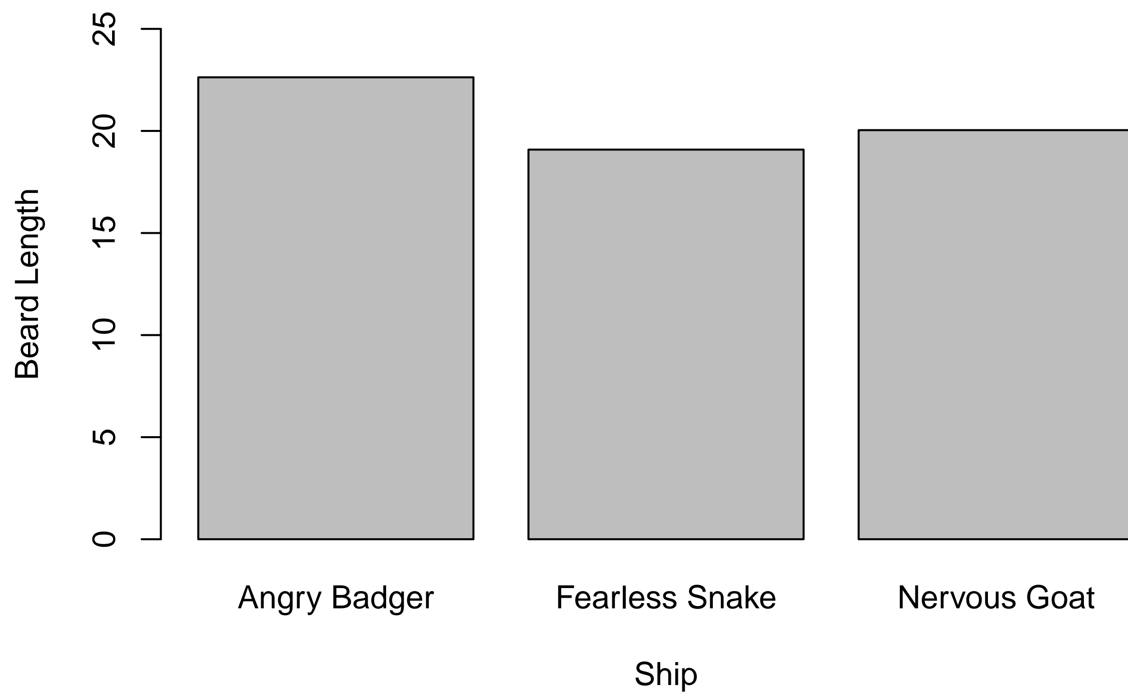


Figure 11.18: Specifying the size and color of points with a third variable.

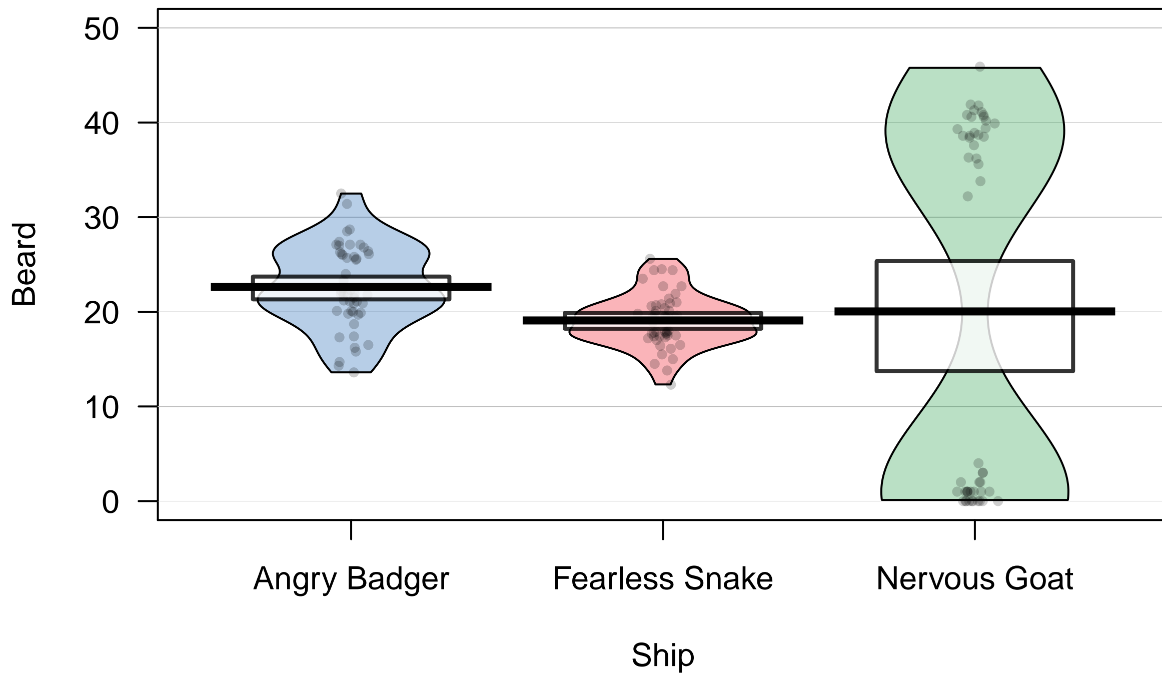
```
grid()
```

11.10 Test your R might! Purdy pictures

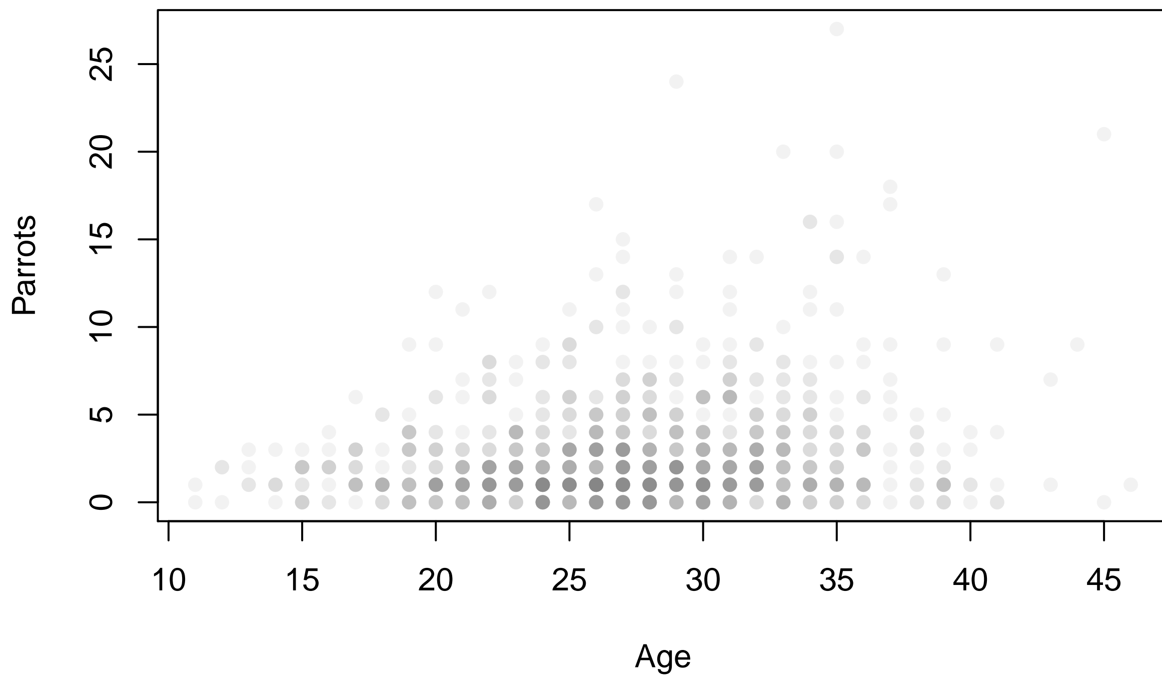
1. The `BeardLengths` dataframe (contained in the `yarrr` package or online at https://github.com/ndphillips/ThePiratesGuideToR/raw/master/data/BeardLengths.txt) contains data on the lengths of beards from 3 different pirate ships. Calculate the average beard length for each ship using `aggregate()`, then create the following barplot:



2. Now using the entire `BeardLengths` dataframe, create the following pirateplot:



3. Using the `pirates` dataset, create the following scatterplot showing the relationship between a pirate's age and how many parrot's (s)he has owned (hint: to make the points solid and transparent, use `pch = 16`, and `col = gray(level = .5, alpha = .1)`).



Chapter 12

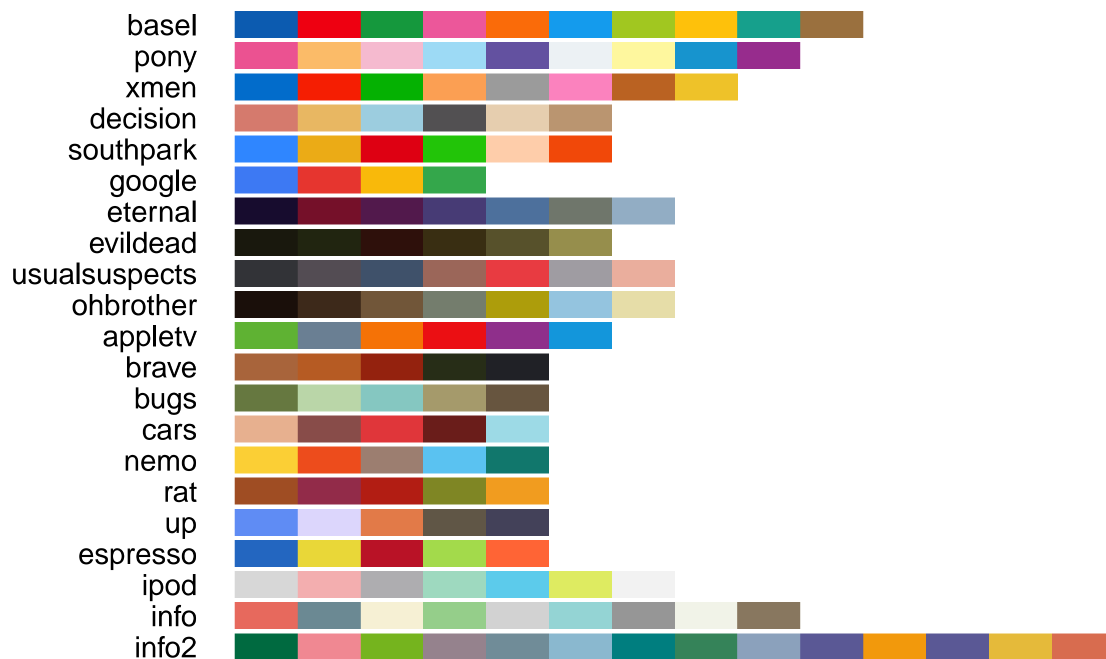
Plotting (II)

12.1 More colors

12.1.1 piratepal()

The `yarrr` package comes with several color palettes ready for you to use. The palettes are contained in the `piratepal()` function. To see all the palettes, run `piratepal("all")`

```
yarrr::piratepal("all")
```

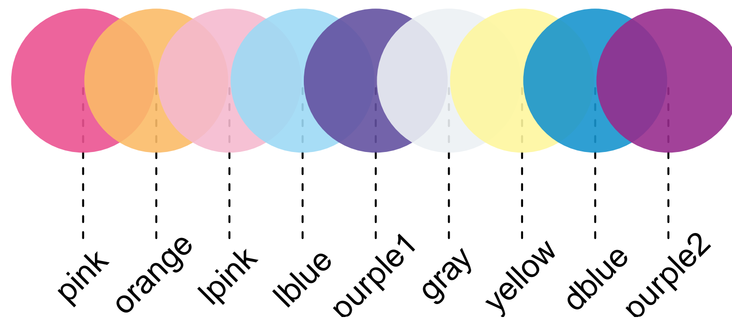


To see a palette in detail, including a picture of what inspired the palette, include the name of the palette in the first argument, (e.g.; `"basel"`) and then specify the argument `plot.result = TRUE`. Here are a few of my personal favorite palettes:

```
# Show me the basel palette
yarrr::piratepal("basel",
                 plot.result = TRUE,
                 trans = .1)          # Slightly transparent
```

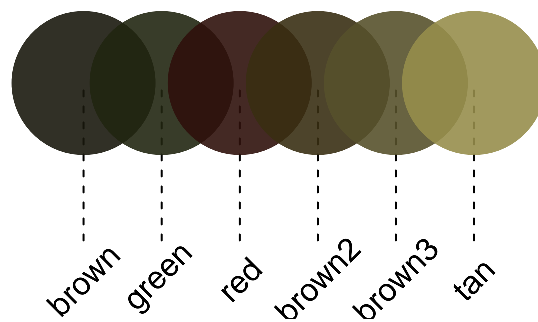
```
# Show me the pony palette
yarrr::piratepal("pony",
                 plot.result = TRUE,
                 trans = .1)          # Slightly transparent
```

pony
trans = 0.1



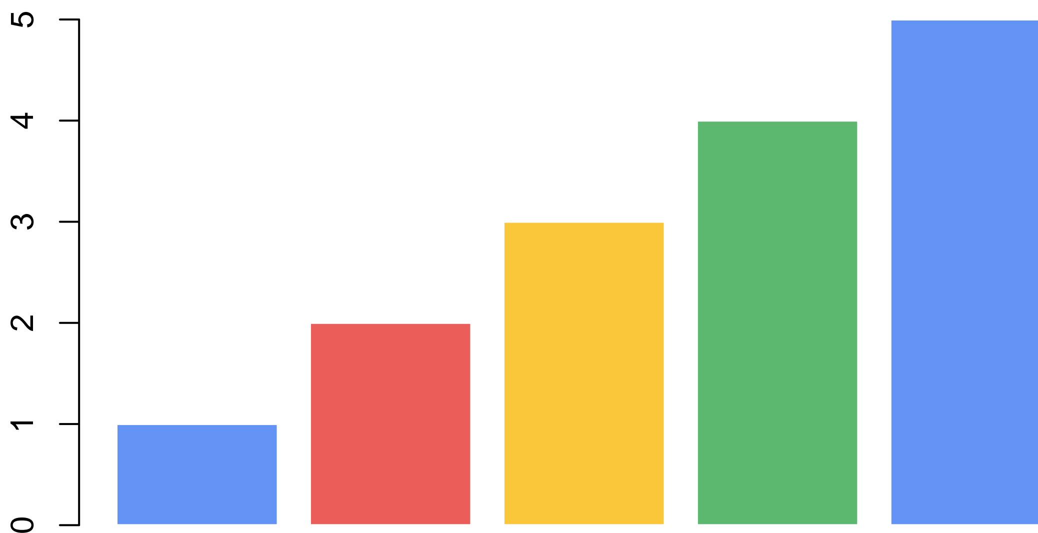

```
# Show me the evildead palette
yarrr::piratepal("evildead",
                 plot.result = TRUE,
                 trans = .1)          # Slightly transparent
```

evildead
trans = 0.1



Once you find a color palette you like, you can save the colors as a vector and assigning the result to an object. For example, if I want to use the "google" palette and use them in a barplot, I would do the following:

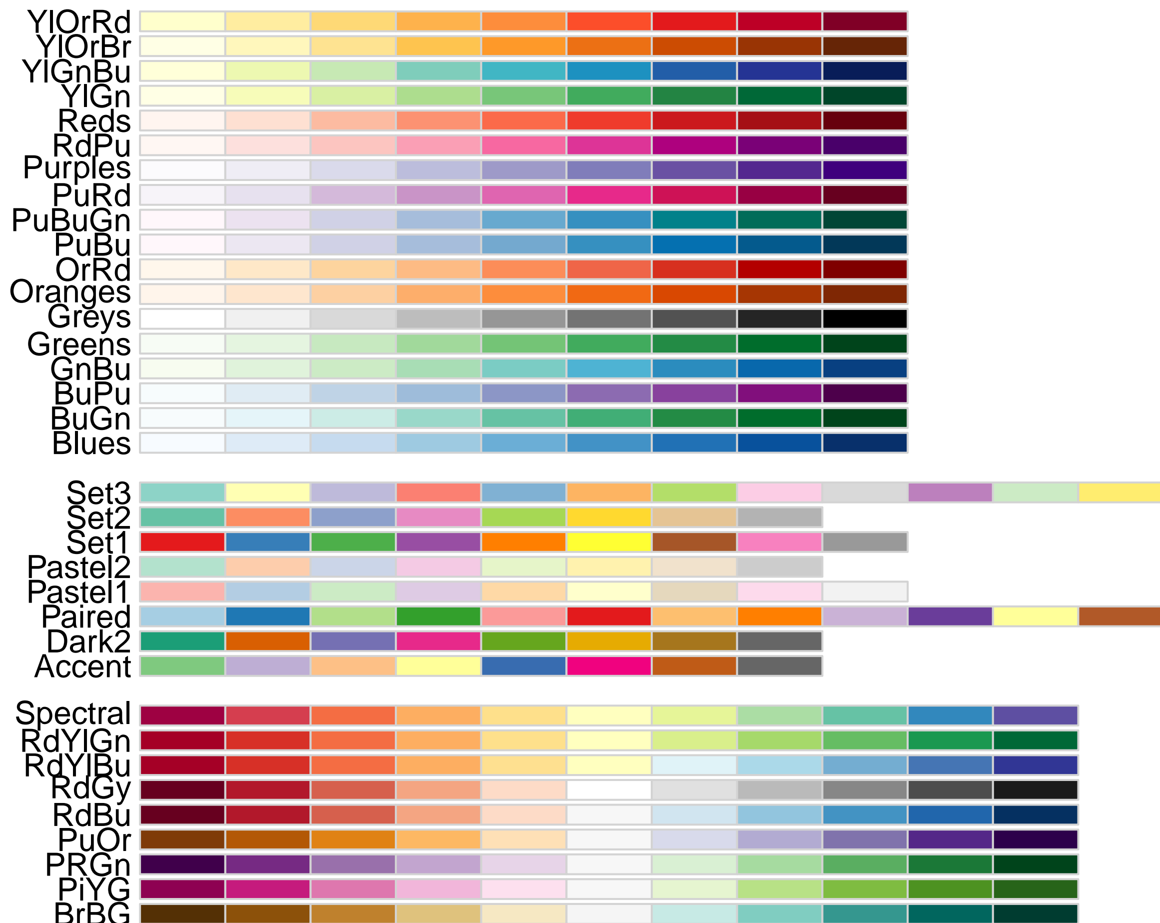
```
# Save the South Park palette to a vector
google.cols <- piratepal(palette = "google",
                         trans = .2)

# Create a barplot with the google colors
barplot(height = 1:5,
       col = google.cols,
       border = "white",
       main = "Barplot with the google palette")
```



12.1.2 RColorBrewer

One package that is great for getting (and even creating) palettes is RColorBrewer. Here are some of the palettes in the package. The name of each palette is in the first column, and the colors in each palette are in each row:

```
library("RColorBrewer")
display.brewer.all()
```



12.1.3 colorRamp2

My favorite way to generate colors that represent numerical data is with the function `colorRamp2` in the `circlize` package (the same package that creates that really cool `chordDiagram` from Chapter 1). The `colorRamp2` function allows you to easily generate shades of colors based on numerical data.

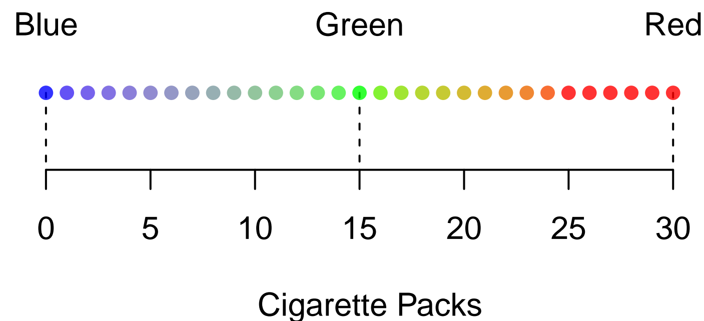
The best way to explain how `colorRamp2` works is by giving you an example. Let's say that you want to want to plot data showing the relationship between the number of drinks someone has on average per week and the resulting risk of some adverse health effect. Further, let's say you want to color the points as a function of the number of packs of cigarettes per week that person smokes, where a value of 0 packs is colored Blue, 10 packs is Orange, and 30 packs is Red. Moreover, you want the values in between these *break points* of 0, 10 and 30 to be a mix of the colors. For example, the value of 5 (half way between 0 and 10) should be an equal mix of Blue and Orange.

When you run the function, the function will actually *return* another function that you can then use to generate colors. Once you store the resulting function as an object (something like `my.color.fun` You can then apply this new function on numerical data (in our example, the number of cigarettes someone smokes) to obtain the correct color for each data point.

For example, let's create the color ramp function for our smoking data points. I'll use `colorRamp2` to create a function that I'll call `smoking.colors` which takes a number as an argument, and returns the corresponding color:

```
# Create color function from colorRamp2
smoking.colors <- circlize::colorRamp2(breaks = c(0, 15, 25),
```

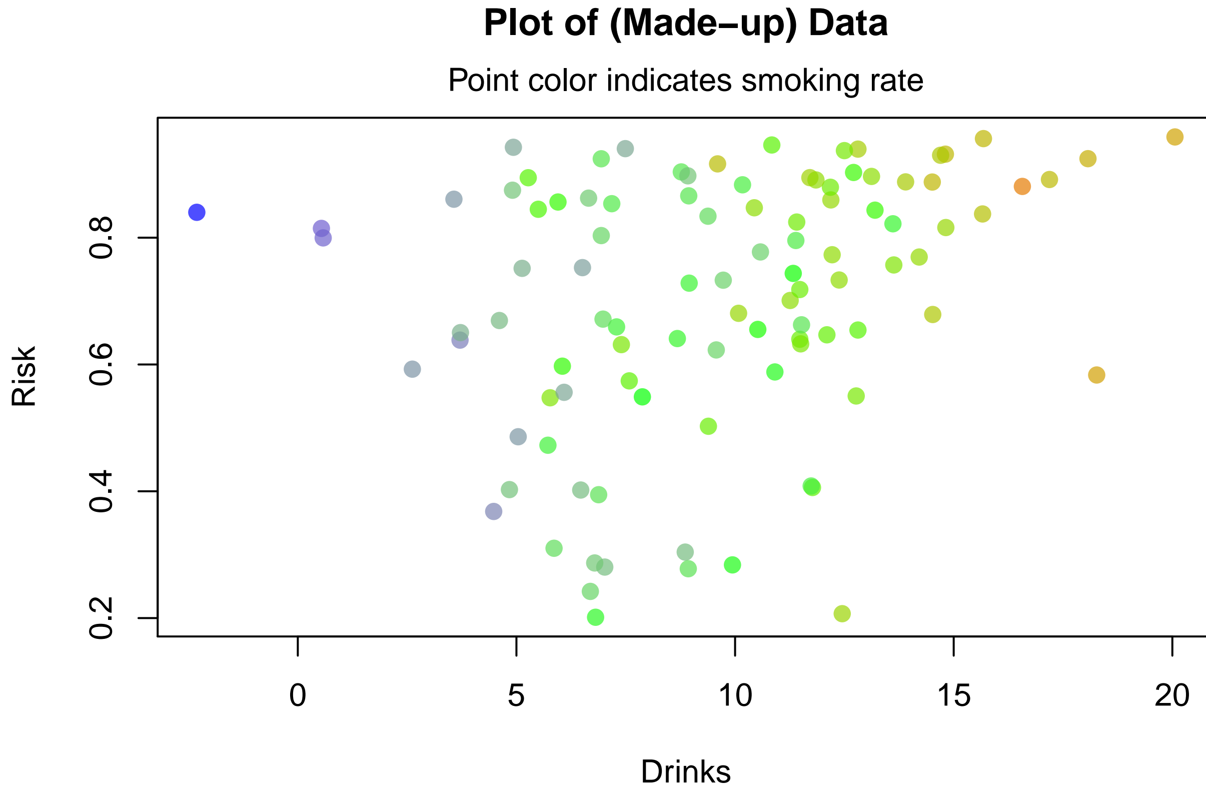
```
                  colors = c("blue", "green", "red"),
                  transparency = .2)


plot(1, xlim = c(-.5, 31.5), ylim = c(0, .3),
     type = "n", xlab = "Cigarette Packs",
     yaxt = "n", ylab = "", bty = "n",
     main = "colorRamp2 Example")

segments(x0 = c(0, 15, 30),
         y0 = rep(0, 3),
         x1 = c(0, 15, 30),
         y1 = rep(.1, 3),
         lty = 2)

points(x = 0:30,
       y = rep(.1, 31), pch = 16,
       col = smoking.colors(0:30))

text(x = c(0, 15, 30), y = rep(.2, 3),
     labels = c("Blue", "Green", "Red"))
```



To see this function in action, check out the the margin Figure~?? for an example, and check out the help menu ?colorRamp2 for more information and examples.

```
# Create Data
drinks <- round(rnorm(100, mean = 10, sd = 4), 2)
smokes <- drinks + rnorm(100, mean = 5, sd = 2)
risk <- 1 / (1 + exp(-(drinks + smokes) / 20 + rnorm(100, mean = 0, sd = 1)))

# Create color function from colorRamp2
smoking.colors <- circlize::colorRamp2(breaks = c(0, 15, 30),
                  colors = c("blue", "green", "red"),
                  transparency = .3)

# Bottom Plot
par(mar = c(4, 4, 5, 1))
plot(x = drinks,
     y = risk,
```

```
     col = smoking.colors(smokes),
     pch = 16, cex = 1.2, main = "Plot of (Made-up) Data",
     xlab = "Drinks", ylab = "Risk")

mtext(text = "Point color indicates smoking rate", line = .5, side = 3)
```



12.1.4 Getting colors with a kuler

One of my favorite tricks for getting great colors in R is to use a *color kuler*. A color kuler is a tool that allows you to determine the exact RGB values for a color on a screen. For example, let's say that you wanted to use the exact colors used in the Google logo. To do this, you need to use an app that allows you to pick colors off your computer screen. On a Mac, you can use the program called "Digital Color Meter." If you then move your mouse over the color you want, the software will tell you the exact RGB values of that color. In the image below, you can see me figuring out that the RGB value of the G in Google is R: 19, G: 72, B: 206. Using the `rgb()` function, I can convert these RGB values to colors in R. Using this method, I figured out the four colors of Google!

```
# Store the colors of google as a vector:
google.col <- c(
  rgb(19, 72, 206, maxColorValue = 255),     # Google blue
  rgb(206, 45, 35, maxColorValue = 255),     # Google red
  rgb(253, 172, 10, maxColorValue = 255),    # Google yellow
  rgb(18, 140, 70, maxColorValue = 255))     # Google green

# Print the result
google.col
## [1] "#1348CE" "#CE2D23" "#FDAC0A" "#128C46"
```

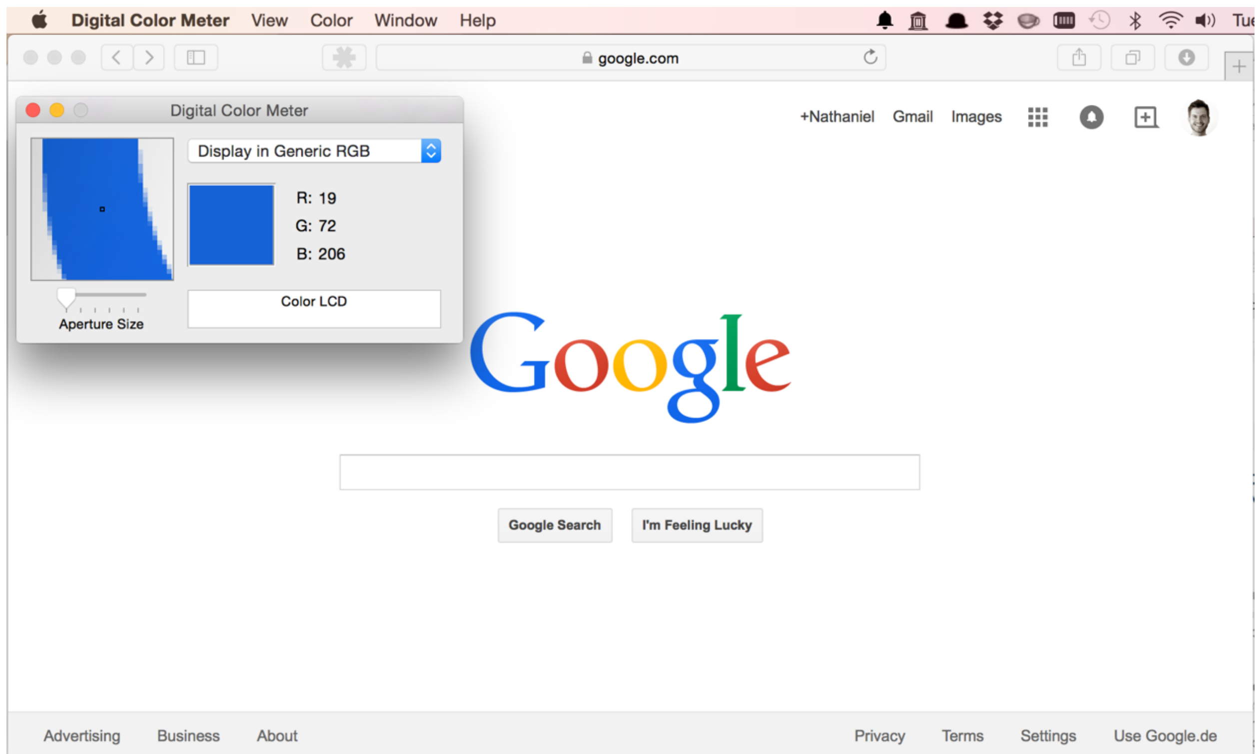


Figure 12.1: Stealing colors from the internet. Not illegal (yet).

The vector `google.col` now contains the values #1348CE, #CE2D23, #FDAC0A, #128C46. These are string values that represent colors in a way R understands. Now I can use these colors in a plot by specifying `col = google.col`!

```
plot(1,
     xlim = c(0, 7),
     ylim = c(0, 1),
     type = "n",
     main = "Using colors stolen from a webpage")

points(x = 1:6,
       y = rep(.4, 6),
       pch = 16,
       col = google.col[c(1, 2, 3, 1, 4, 2)],
       cex = 4)

text(x = 1:6,
     y = rep(.7, 6),
     labels = c("G", "O", "O", "G", "L", "E"),
     col = google.col[c(1, 2, 3, 1, 4, 2)],
     cex = 3)
```

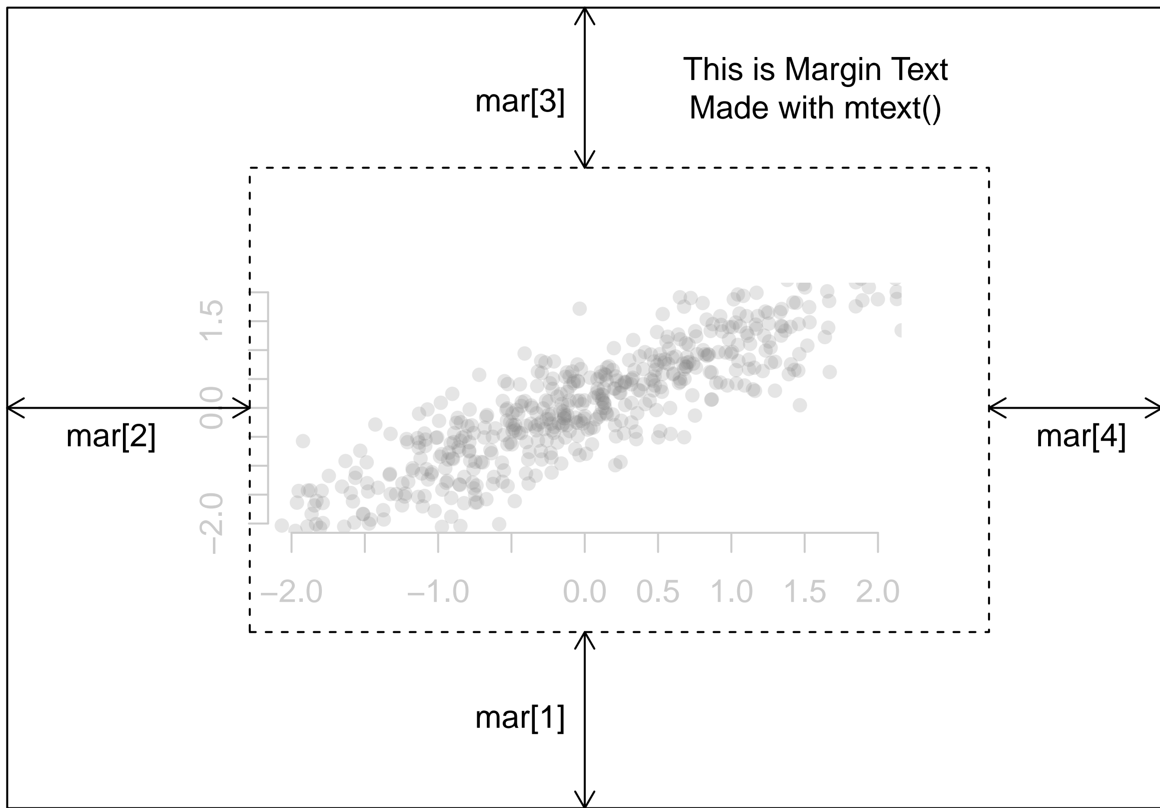
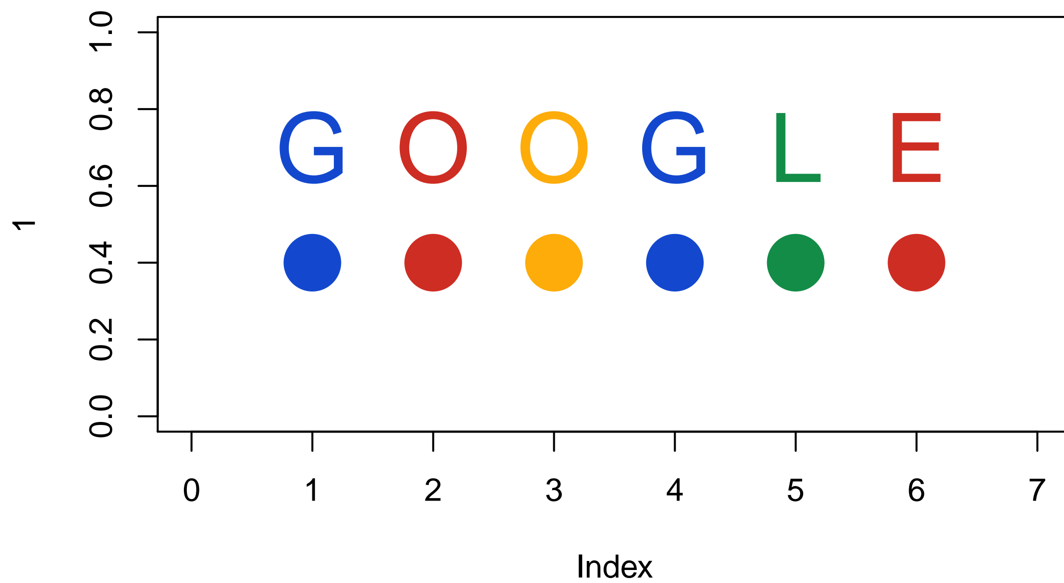


Figure 12.2: Margins of a plot.



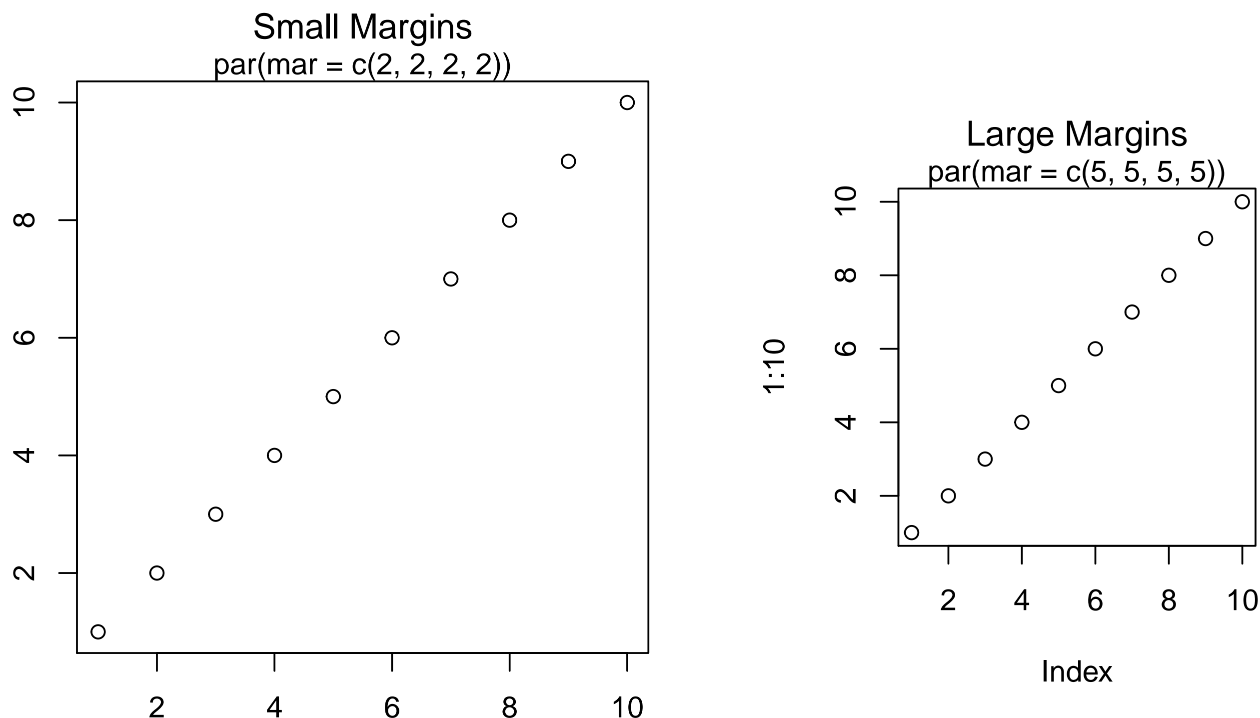
12.2 Plot Margins

All plots in R have margins surrounding them that separate the main plotting space from the area where the axes, labels and additional text lie. To visualize how R creates plot margins, look at margin Figure 12.2.

You can adjust the size of the margins by specifying a margin parameter using the syntax `par(mar = c(bottom, left, top, right))`, where the arguments `bottom`, `left` … are the size of the margins. The default value for `mar` is c(5.1, 4.1, 4.1, 2.1). To change the size of the margins of a plot you must do so with `par(mar)` *before* you actually create the plot.

Let's see how this works by creating two plots with different margins:In the plot on the left, I'll set the margins to 3 on all sides. In the plot on the right, I'll set the margins to 6 on all sides.

```
# First Plot with small margins
par(mar = c(2, 2, 2, 2)) # Set the margin on all sides to 2
plot(1:10)
mtext("Small Margins", side = 3, line = 1, cex = 1.2)
mtext("par(mar = c(2, 2, 2, 2))", side = 3)

# Second Plot with large margins
par(mar = c(5, 5, 5, 5)) # Set the margin on all sides to 6
plot(1:10)
mtext("Large Margins", side = 3, line = 1, cex = 1.2)
mtext("par(mar = c(5, 5, 5, 5))", side = 3)
```



You'll notice that the margins are so small in the first plot that you can't even see the axis labels, while in the second plot there is plenty (probably too much) white space around the plotting region.

In addition to using the `mar` parameter, you can also specify margin sizes with the `mai` parameter. This acts just like `mar` except that the values for `mai` set the margin size in inches.

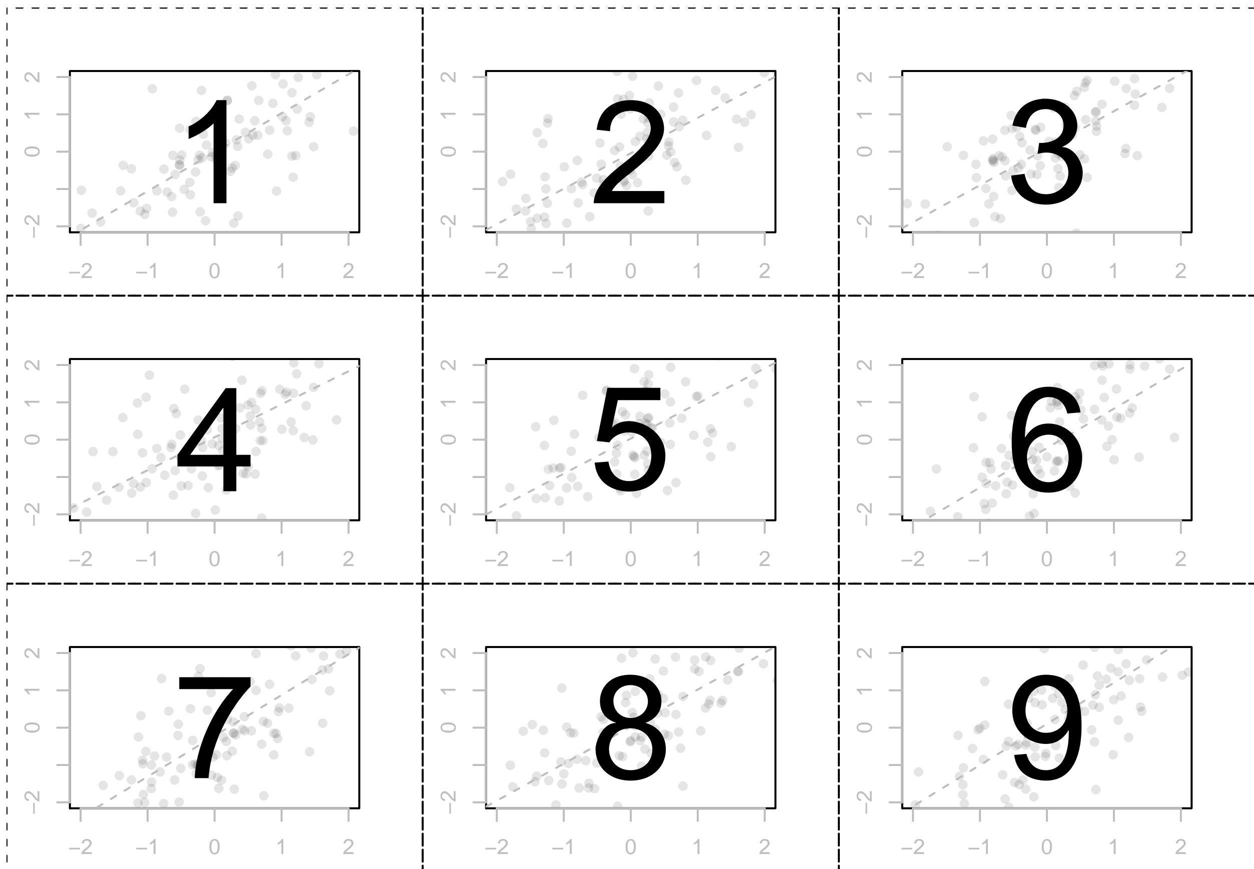


Figure 12.3: A 3 x 3 matrix of plotting regions created by par(mfrow = c(3, 3))

12.3 Arranging plots with `par(mfrow)` and `layout()`

R makes it easy to arrange multiple plots in the same plotting space. The most common ways to do this is with the `par(mfrow)` parameter, and the `layout()` function. Let's go over each in turn:

The `mfrow` and `mfcol` parameters allow you to create a matrix of plots in one plotting space. Both parameters take a vector of length two as an argument, corresponding to the number of rows and columns in the resulting plotting matrix. For example, the following code sets up a 3 x 3 plotting matrix.

```
par(mfrow = c(2, 2)) # Create a 2 x 2 plotting matrix
# The next 4 plots created will be plotted next to each other

# Plot 1
hist(rnorm(100))

# Plot 2
plot(pirates$weight,
     pirates$height, pch = 16, col = gray(.3, .1))

# Plot 3
pirateplot(weight ~ Diet,
           data = ChickWeight,
           pal = "info", theme = 3)
```

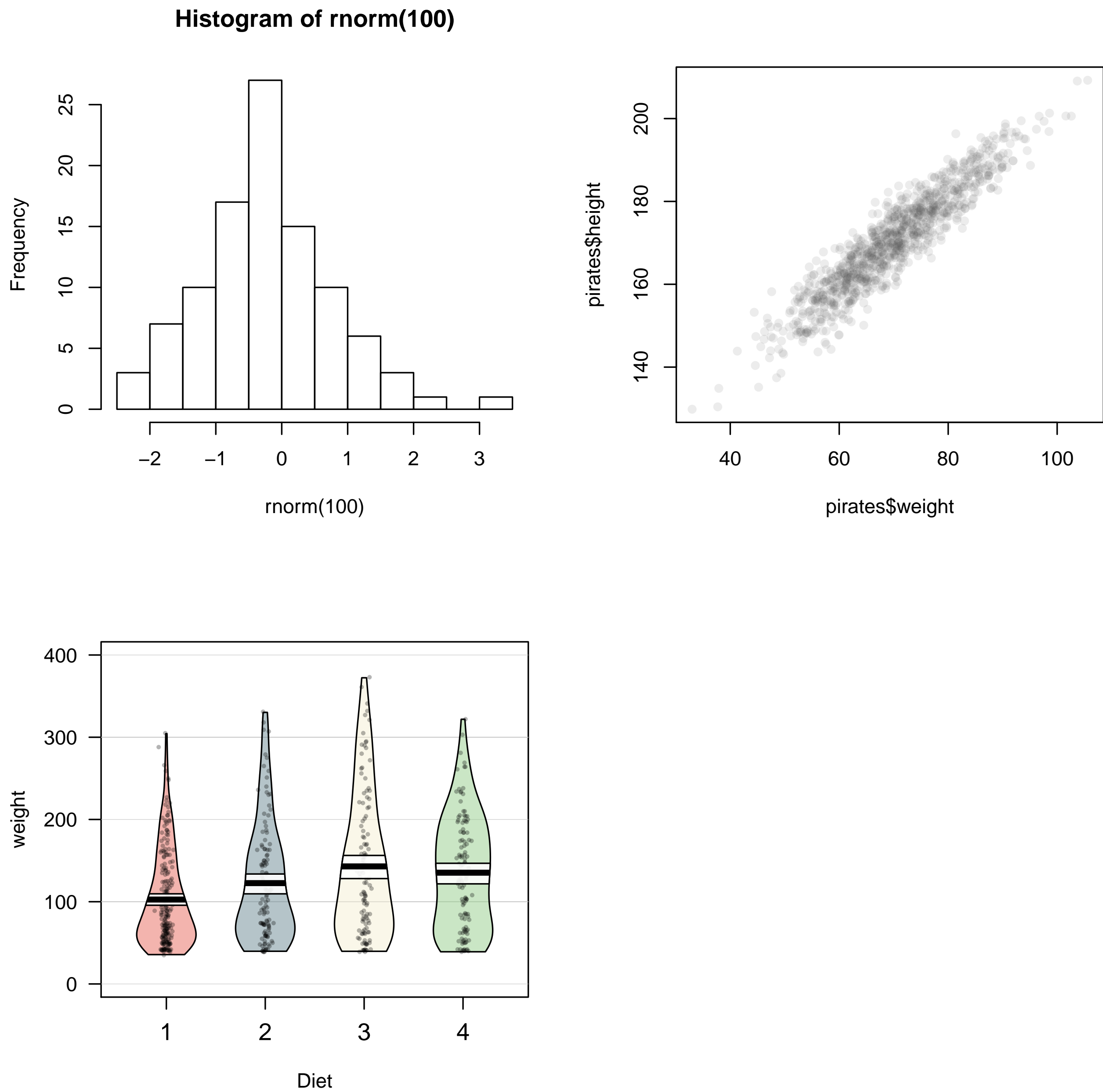


Figure 12.4: Arranging plots into a 2x2 matrix with par(mfrow = c(2, 2))

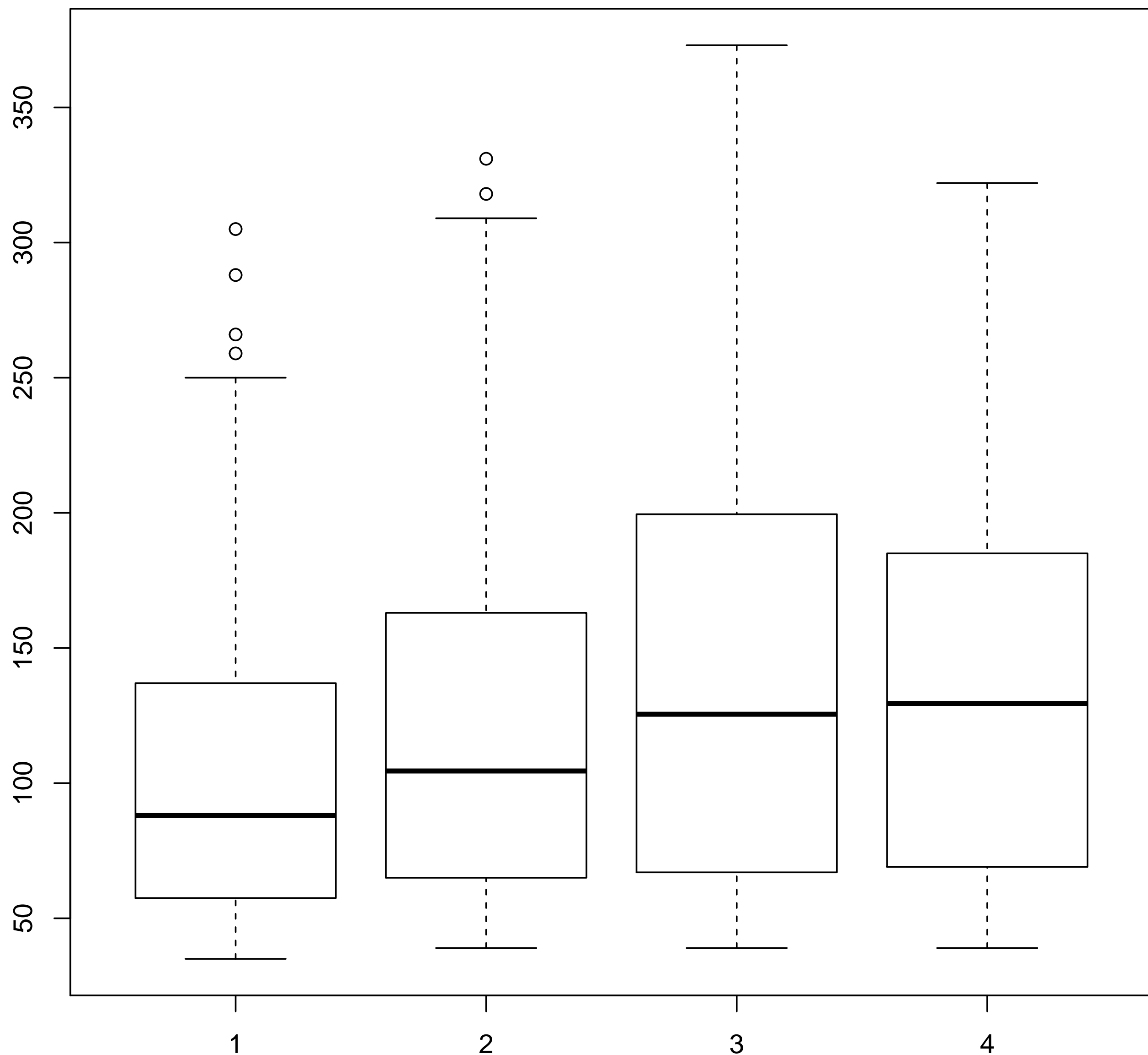


Figure 12.5: Arranging plots into a 2x2 matrix with par(mfrow = c(2, 2))

```
# Plot 4
boxplot(weight ~ Diet,
           data = ChickWeight)
```

When you execute this code, you won't see anything happen. However, when you execute your first high-level plotting command, you'll see that the plot will show up in the space reserved for the first plot (the top left). When you execute a second high-level plotting command, R will place that plot in the second place in the plotting matrix - either the top middle (if using `par(mfrow)` or the left middle (if using `par(mfcol)`). As you continue to add high-level plots, R will continue to fill the plotting matrix.

So what's the difference between `par(mfrow)` and `par(mfcol)`? The only difference is that while `par(mfrow)` puts sequential plots into the plotting matrix by row, `par(mfcol)` will fill them by column.

When you are finished using a plotting matrix, be sure to reset the plotting parameter back to its default state by running `par(mfrow = c(1, 1))`:

```
# Put plotting arrangement back to its original state
par(mfrow = c(1, 1))
```

12.3.1 Complex plot layouts with `layout()`

Argument	Description
`mat`	A matrix indicating the location of the next N figures in the global plotting space. Each value in the matrix must be 0 or a positive integer. R will plot the first plot in the entries of the matrix with 1, the second plot in the entries with 2,…
`widths`	A vector of values for the widths of the columns of the plotting space.
`heights`	A vector of values for the heights of the rows of the plotting space.

While `par(mfrow)` allows you to create matrices of plots, it does not allow you to create plots of different sizes. In order to arrange plots in different sized plotting spaces, you need to use the `layout()` function. Unlike `par(mfrow)`, `layout` is not a plotting parameter, rather it is a function all on its own. The function can be a bit confusing at first, so I think it's best to start with an example. Let's say you want to place histograms next to a scatterplot: Let's do this using layout:

We'll begin by creating the *layout matrix*, this matrix will tell R in which order to create the plots:

```
layout.matrix <- matrix(c(0, 2, 3, 1), nrow = 2, ncol = 2)
layout.matrix
##      [,1] [,2]
## [1,]    0    3
## [2,]    2    1
```

Looking at the values of `layout.matrix`, you can see that we've told R to put the first plot in the bottom right, the second plot on the bottom left, and the third plot in the top right. Because we put a 0 in the first element, R knows that we don't plan to put anything in the top left area.

Now, because our layout matrix has two rows and two columns, we need to set the widths and heights of the two columns. We do this using a numeric vector of length 2. I'll set the heights of the two rows to 1 and 2 respectively, and the widths of the columns to 1 and 2 respectively. Now, when I run the code `layout.show(3)`, R will show us the plotting region we set up:

```
layout.matrix <- matrix(c(2, 1, 0, 3), nrow = 2, ncol = 2)

layout(mat = layout.matrix,
       heights = c(1, 2), # Heights of the two rows
       widths = c(2, 2)) # Widths of the two columns

layout.show(3)
```

Now we're ready to put the plots together

```
# Set plot layout
layout(mat = matrix(c(2, 1, 0, 3),
                        nrow = 2,
                        ncol = 2),
       heights = c(1, 2),    # Heights of the two rows
       widths = c(2, 1))     # Widths of the two columns
```

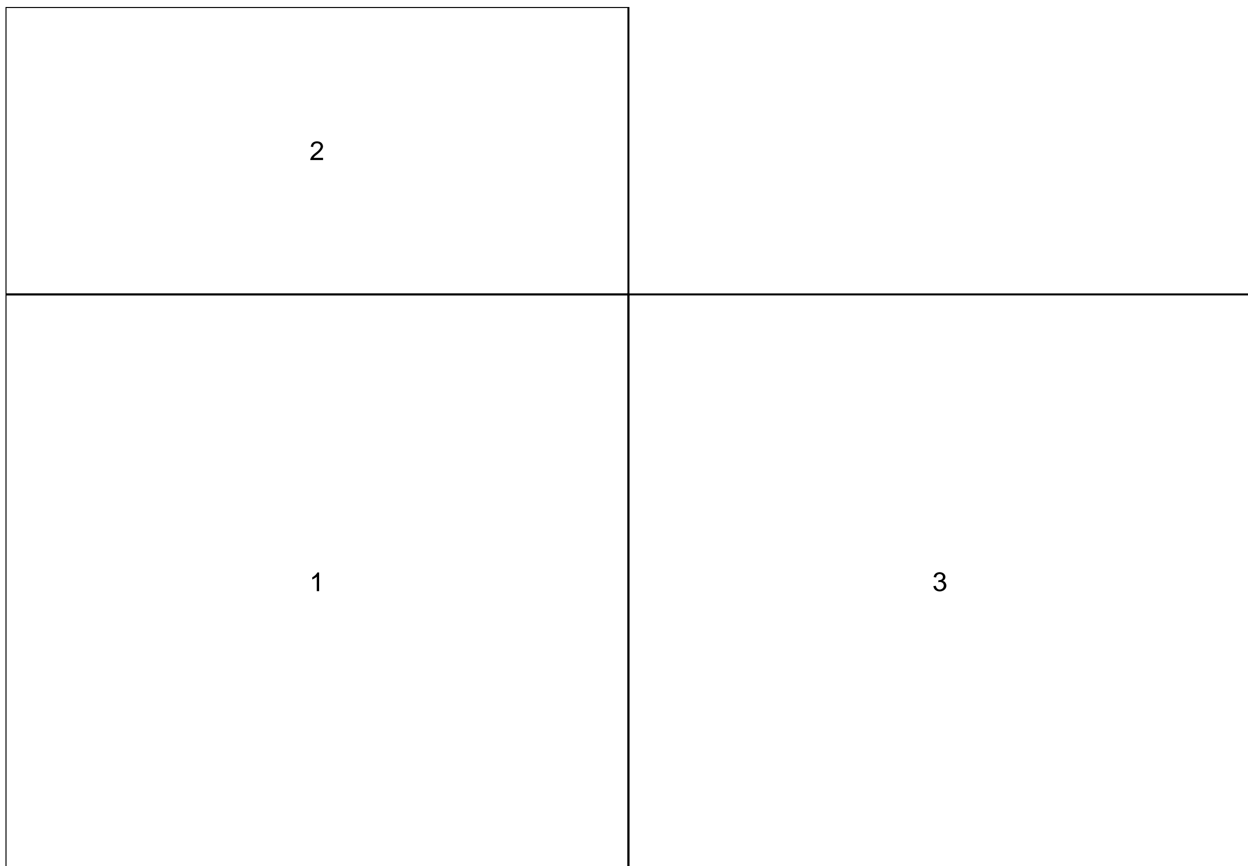


Figure 12.6: A plotting layout created by setting a layout matrix with two rows and two columns. The first row has a height of 1, and the second row has a hight of 2. Both columns have the same width of 2.

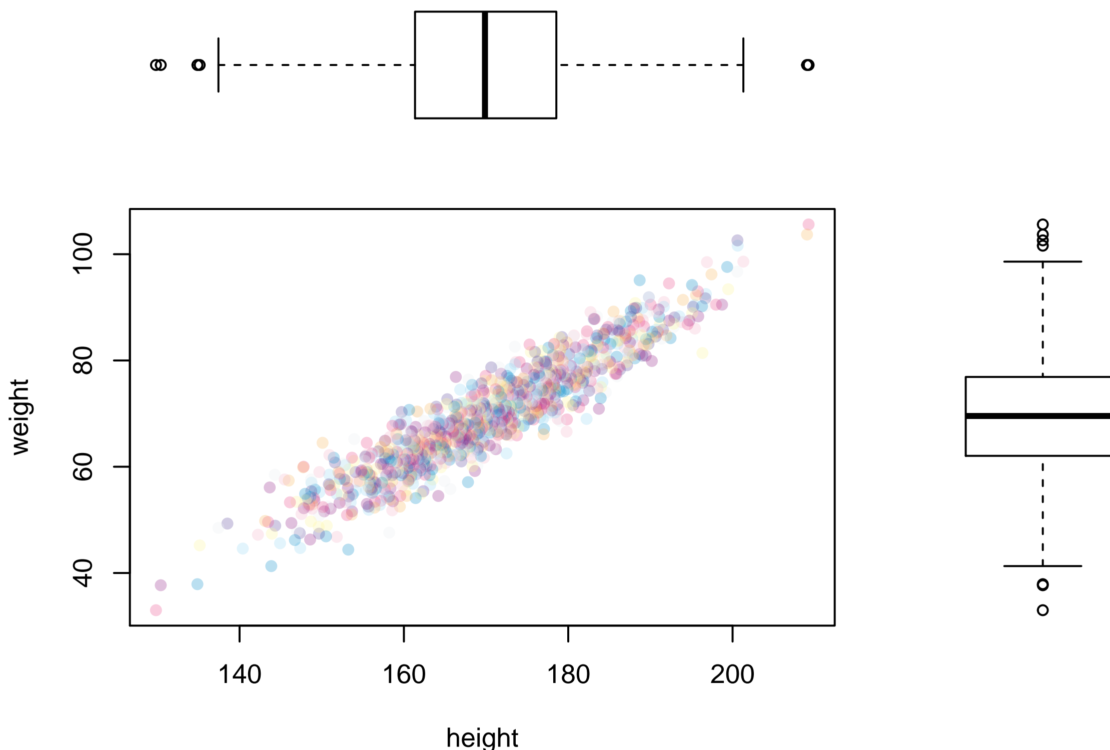


Figure 12.7: Adding boxplots to margins of a scatterplot with layout().

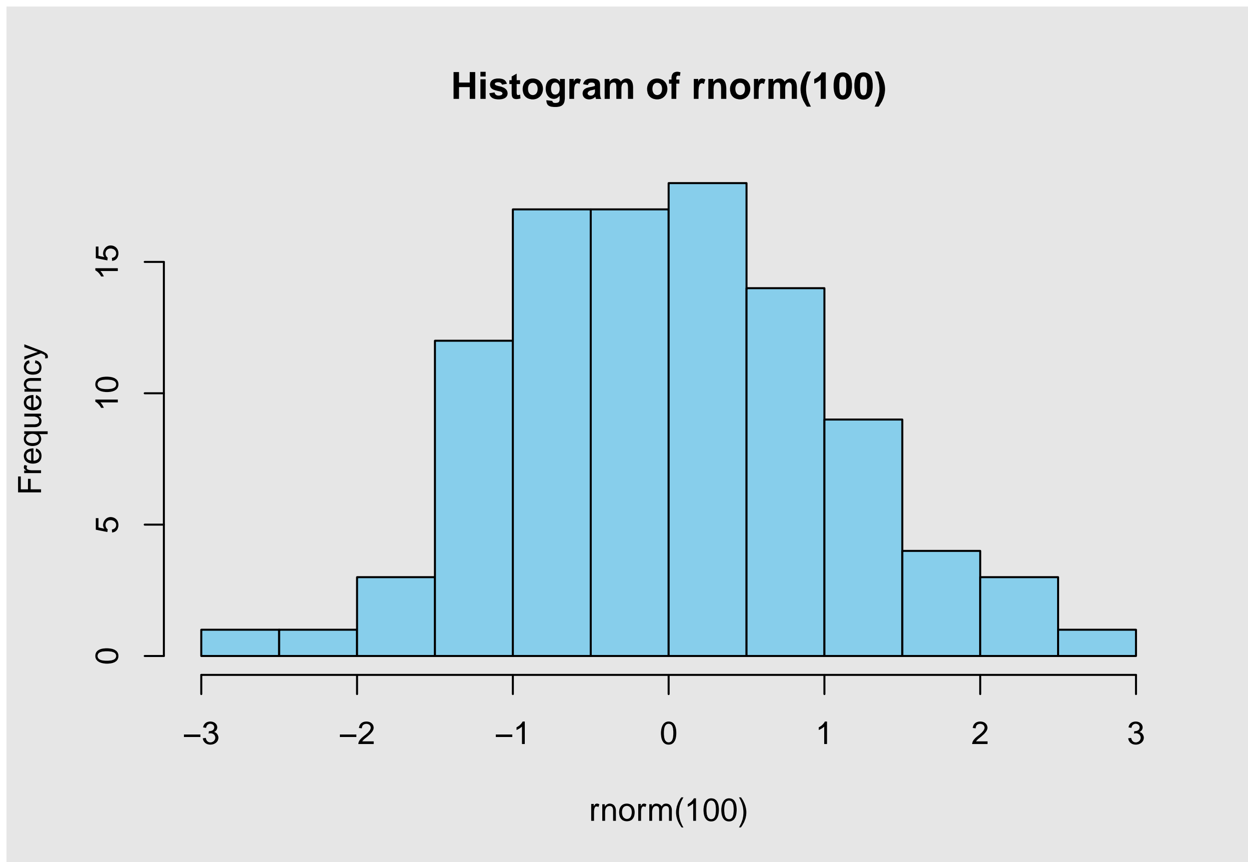
```
# Plot 1: Scatterplot
par(mar = c(5, 4, 0, 0))
plot(x = pirates$height,
     y = pirates$weight,
     xlab = "height",
     ylab = "weight",
     pch = 16,
     col = yarrr::piratepal("pony", trans = .7))

# Plot 2: Top (height) boxplot
par(mar = c(0, 4, 0, 0))
boxplot(pirates$height, xaxt = "n",
        yaxt = "n", bty = "n", yaxt = "n",
        col = "white", frame = FALSE, horizontal = TRUE)

# Plot 3: Right (weight) boxplot
par(mar = c(5, 0, 0, 0))
boxplot(pirates$weight, xaxt = "n",
        yaxt = "n", bty = "n", yaxt = "n",
        col = "white", frame = F)
```

12.4 Additional plotting parameters

To change the background color of a plot, add the command `par(bg = col)` (where `col` is the color you want to use) prior to creating the plot. For example, the following code will put a light gray background

behind a histogram:

```
par(bg = gray(.9)) # Create a light gray background
hist(x = rnorm(100), col = "skyblue")
```



Here's a more complex example:

```
parrot.data <- data.frame(
  "ship" = c("Drunken\nMonkeys", "Slippery\nSnails", "Don't Ask\nDon't Tell", "The Beliebers"),
  "Green" = c(200, 150, 100, 175),
  "Blue " = c(150, 125, 180, 242))

# Set background color and plot margins
par(bg = rgb(61, 55, 72, maxColorValue = 255),
    mar = c(6, 6, 4, 3))

plot(1, xlab = "", ylab = "", xaxt = "n",
     yaxt = "n", main = "", bty = "n", type = "n",
     ylim = c(0, 250), xlim = c(.25, 5.25))

# Add gridlines
abline(h = seq(0, 250, 50),
       lty = 3,
       col = gray(.95), lwd = 1)

# y-axis labels
mtext(text = seq(50, 250, 50),
```

```
      side = 2, at = seq(50, 250, 50),
      las = 1, line = 1, col = gray(.95))

# ship labels
mtext(text = parrot.data$ship,
      side = 1, at = 1:4, las = 1,
      line = 1, col = gray(.95))

# Blue bars
rect(xleft = 1:4 - .35 - .04 / 2,
     ybottom = rep(0, 4),
     xright = 1:4 - .04 / 2,
     ytop = parrot.data$Blue,
     col = "lightskyblue1", border = NA)

# Green bars
rect(xleft = 1:4 + .04 / 2,
     ybottom = rep(0, 4),
     xright = 1:4 + .35 + .04 / 2,
     ytop = parrot.data$Green,
     col = "lightgreen", border = NA)

legend(4.5, 250, c("Blue", "Green"),
       col = c("lightskyblue1", "lightgreen"), pch = rep(15, 2),
       bty = "n", pt.cex = 1.5, text.col = "white")

# Additional margin text
mtext("Number of Green and Blue parrots on 4 ships",
      side = 3, cex = 1.5, col = "white")
mtext("Parrots", side = 2, col = "white", line = 3.5)
mtext("Source: Drunken survey on 22 May 2015", side = 1,
      at = 0, adj = 0, line = 3, font = 3, col = "white")
```

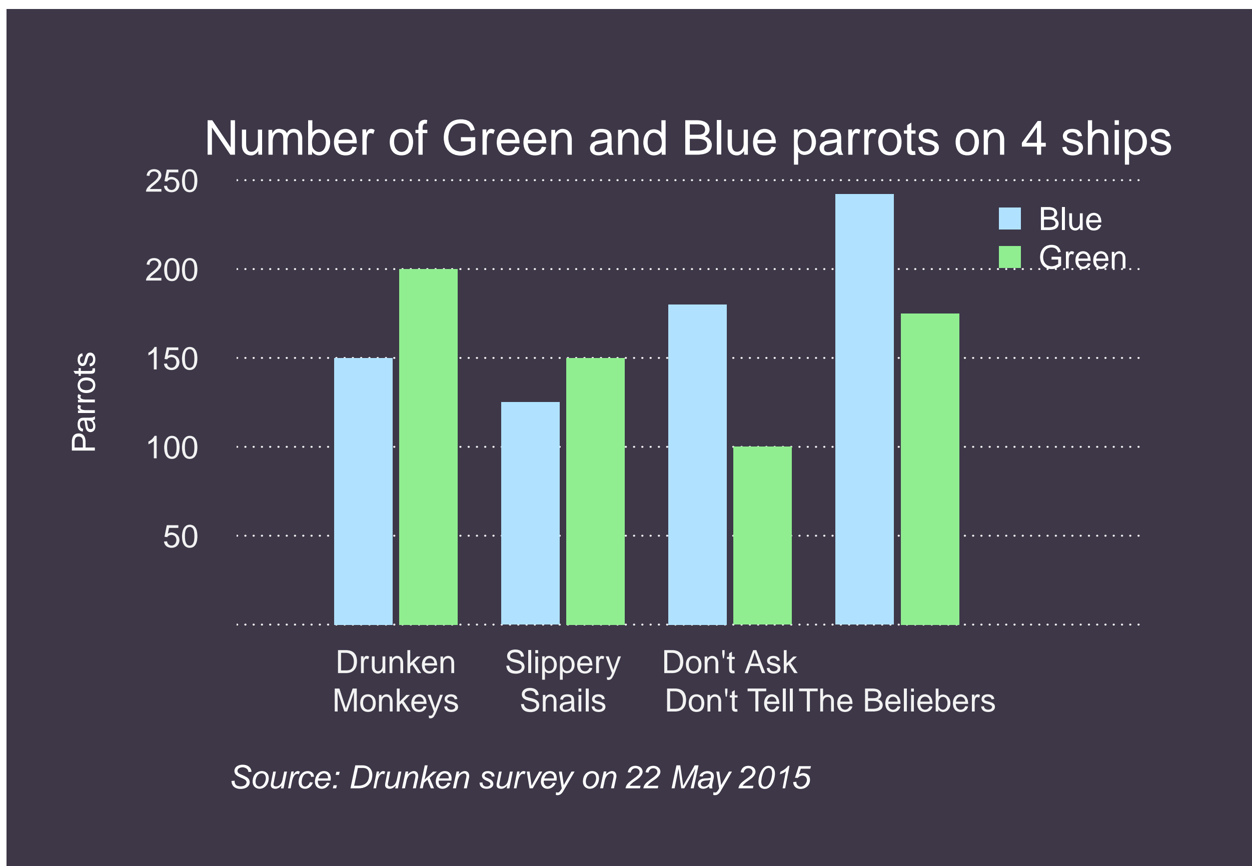


Figure 12.8: Use par(bg = my.color) before creating a plot to add a colored background.

Chapter 13

Hypothesis Tests

In this chapter we'll cover 1 and 2 sample null hypothesis tests: like the t-test, correlation test, and chi-square test:

```
library(yarrr) # Load yarrr to get the pirates data

# 1 sample t-test
# Are pirate ages different than 30 on average?
t.test(x = pirates$age,
       mu = 30)

# 2 sample t-test
# Do females and males have different numbers of  tattoos?
sex.ttest <- t.test(formula = tattoos ~ sex,
                    data = pirates,
                    subset = sex %in% c("male", "female"))
sex.ttest # Print result

## Access specific values from test
sex.ttest$statistic
sex.ttest$p.value
sex.ttest$conf.int

# Correlation test
# Is there a relationship between age and height?
 cor.test(formula = ~ age + height,
          data = pirates)

# Chi-Square test
# Is there a relationship between college and favorite pirate?
chisq.test(x = pirates$college,
           y = pirates$favorite.pirate)
```

Do we get more treasure from chests buried in sand or at the bottom of the ocean? Is there a relationship between the number of scars a pirate has and how much grogg he can drink? Are pirates with body piercings more likely to wear bandannas than those without body piercings? Glad you asked, in this chapter, we'll answer these questions using 1 and 2 sample frequentist hypothesis tests.

As this is a Pirate's Guide to R, and not a Pirate's Guide to Statistics, we won't cover all the theoretical background behind frequentist null hypothesis tests (like t-tests) in much detail. However, it's important to

Figure 13.1: Sadly, this still counts as just one tattoo.

cover three main concepts: Descriptive statistics, Test statistics, and p-values. To do this, let's talk about body piercings.

13.1 A short introduction to hypothesis tests

As you may know, pirates are quite fond of body piercings. Both as a fashion statement, and as a handy place to hang their laundry. Now, there is a stereotype that European pirates have more body piercings than American pirates. But is this true? To answer this, I conducted a survey where I asked 10 American and 10 European pirates how many body piercings they had. The results are below, and a Pirateplot of the data is in Figure **??**:

```
# Body piercing data
american.bp <- c(3, 5, 2, 1, 4, 4, 6, 3, 5, 4)
european.bp <- c(6, 5, 7, 7, 6, 3, 4, 6, 5, 4)

# Store data in a dataframe
bp.survey <- data.frame("bp" = c(american.bp, european.bp),
                        "group" = rep(c("American", "European"), each = 10),
                         stringsAsFactors = FALSE)
```

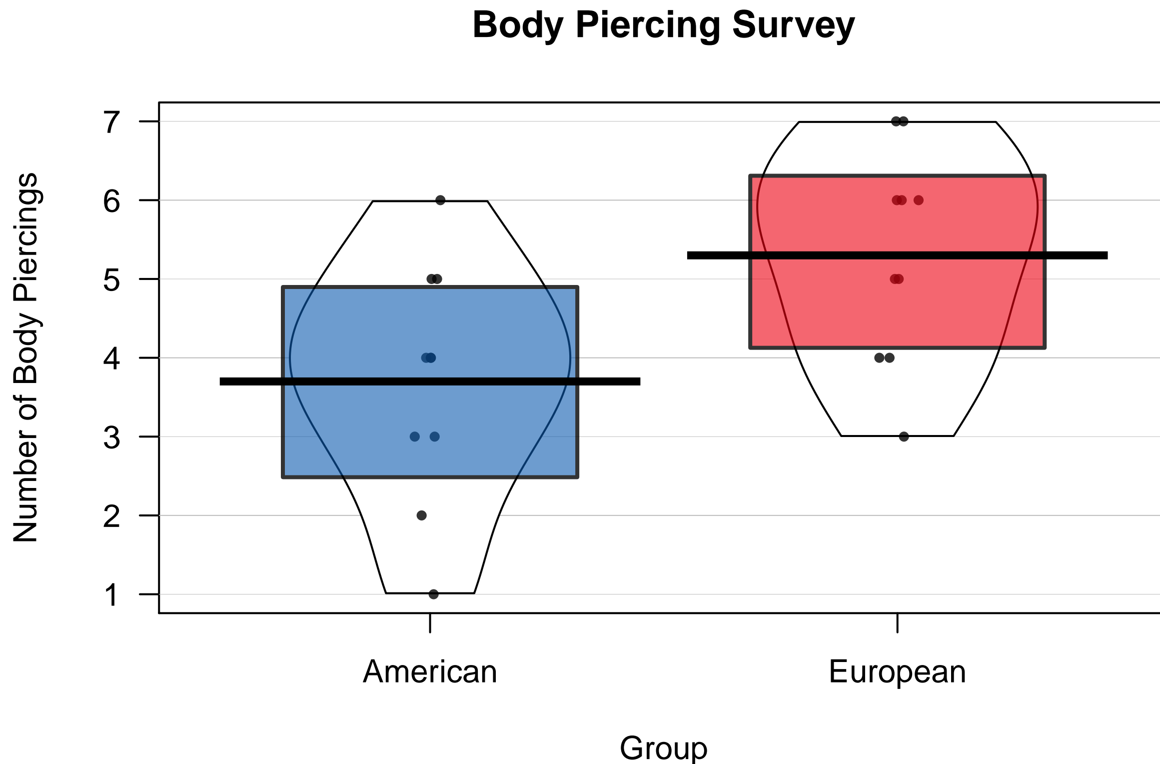


Figure 13.2: A Pirateplot of the body piercing data.

```
yarrr::pirateplot(bp ~ group,
                  data = bp.survey,
                  main = "Body Piercing Survey",
                  ylab = "Number of Body Piercings",
                  xlab = "Group",
                  theme = 2, point.o = .8, cap.beans = TRUE)
```

13.1.1 Null v Alternative Hypothesis

In null hypothesis tests, you always start with a *null hypothesis*. The specific null hypothesis you choose will depend on the type of question you are asking, but in general, the null hypothesis states that *nothing is going on and everything is the same*. For example, in our body piercing study, our null hypothesis is that American and European pirates have the *same* number of body piercings on average.

The *alternative* hypothesis is the opposite of the null hypothesis. In this case, our alternative hypothesis is that American and European pirates do *not* have the same number of piercings on average. We can have different types of alternative hypotheses depending on how specific we want to be about our prediction. We can make a *1-sided* (also called 1-tailed) hypothesis, by predicting the *direction* of the difference between American and European pirates. For example, our alternative hypothesis could be that European pirates have *more* piercings on average than American pirates.

Alternatively, we could make a *2-sided* (also called 2-tailed) alternative hypothesis that American and European pirates simply differ in their average number of piercings, without stating which group has more piercings than the other.

Once we've stated our null and alternative hypotheses, we collect data and then calculate *descriptive*

statistics.

13.1.2 Descriptive statistics

Descriptive statistics (also called sample statistics) describe samples of data. For example, a mean, median, or standard deviation of a dataset is a descriptive statistic of that dataset. Let's calculate some descriptive statistics on our body piercing survey American and European pirates using the `summary()` function:

```
# Pring descriptive statistics of the piercing data
summary(american.bp)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0     3.0     4.0     3.7     4.8     6.0
summary(european.bp)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     3.0     4.2     5.5     5.3     6.0     7.0
```

Well, it looks like our sample of 10 American pirates had 3.7 body piercings on average, while our sample of 10 European pirates had 5.3 piercings on average. But is this difference large or small? Are we justified in concluding that American and European pirates *in general* differ in how many body piercings they have? To answer this, we need to calculate a *test statistic*

13.1.3 Test Statistics

An test statistic compares descriptive statistics, and determines how different they are. The formula you use to calculate a test statistics depends the type of test you are conducting, which depends on many factors, from the scale of the data (i.e.; is it nominal or interval?), to how it was collected (i.e.; was the data collected from the same person over time or were they all different people?), to how its distributed (i.e.; is it bell-shaped or highly skewed?).

For now, I can tell you that the type of data we are analyzing calls for a two-sample T-test. This test will take the descriptive statistics from our study, and return a test-statistic we can then use to make a decision about whether American and European pirates really differ. To calculate a test statistic from a two-sample t-test, we can use the `t.test()` function in R. Don't worry if it's confusing for now, we'll go through it in detail shortly.

```
# Conduct a two-sided t-test comparing the vectors american.bp and european.bp
#  and save the results in an object called bp.test
bp.test <- t.test(x = american.bp,
                  y = european.bp,
                  alternative = "two.sided")

# Print the main results
bp.test
##
##  Welch Two Sample t-test
##
## data:  american.bp and european.bp
## t = -3, df = 20, p-value = 0.02
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.93 -0.27
## sample estimates:
## mean of x mean of y
##       3.7       5.3
```

Figure 13.3: p-values are like bullshit detectors against the null hypothesis. The smaller the p-value, the more likely it is that the null-hypothesis (the idea that the groups are the same) is bullshit.

It looks like our test-statistic is -2.52. If there was really no difference between the groups of pirates, we would expect a test statistic close to 0. Because test-statistic is -2.52, this makes us think that there really is a difference. However, in order to make our decision, we need to get the *p-value* from the test.

13.1.4 p-value

The p-value is a probability that reflects how consistent the test statistic is *with the hypothesis that the groups are actually the same*. Or more formally, a p-value can be interpreted as follows:

13.1.4.1 Definition of a p-value: Assuming that there the null hypothesis is true (i.e.; that there is no difference between the groups), what is the probability that we would have gotten a test statistic as far away from 0 as the one we actually got?

For this problem, we can access the p-value as follows:

```
# What is the p-value from our t-test?
bp.test$p.value
## [1] 0.021
```

The p-value we got was 0.02, this means that, assuming the two populations of American and European pirates have the same number of body piercings on average, the probability that we would obtain a test statistic as large as -2.52 is 2.1% . This is very small, but is it small enough to decide that the null hypothesis is not true? It's hard to say and there is no definitive answer. However, most pirates use a decision threshold of $p < 0.05$ to determine if we should reject the null hypothesis or not. In other words, if you obtain a p-value less than 0.05, then you reject the null hypothesis. Because our p-value of 0.02 is less than 0.05, we would reject the null hypothesis and conclude that the two populations are *not* be the same.

13.1.4.2 p-values are bullshit detectors against the null hypothesis

P-values sounds complicated – because they are (In fact, most psychology PhDs get the definition wrong). It's very easy to get confused and not know what they are or how to use them. But let me help by putting it another way: a p-value is like a bullshit detector *against* the null hypothesis that goes off when the p-value is too small. If a p-value is too small, the bullshit detector goes off and says "Bullshit! There's no way you would get data like that if the groups were the same!" If a p-value is not too small, the bullshit alarm stays silent, and we conclude that we cannot reject the null hypothesis.

Figure 13.4: Despite what you may see in movies, men cannot get pregnant. And despite what you may want to believe, p-values do not tell you the probability that the null hypothesis is true!

13.1.4.3 How small of a p-value is too small?

Traditionally a p-value of 0.05 (or sometimes 0.01) is used to determine 'statistical significance.' In other words, if a p-value is less than .05, most researchers then conclude that the null hypothesis is false. However, .05 is not a magical number. Anyone who really believes that a p-value of .06 is *much* less significant than a p-value of 0.04 has been sniffing too much glue. However, in order to be consistent with tradition, I will adopt this threshold for the remainder of this chapter. That said, let me reiterate that a p-value threshold of 0.05 is just as arbitrary as a p-value of 0.09, 0.06, or 0.12156325234.

13.1.4.4 Does the p-value tell us the probability that the null hypothesis is true?

No!!! The p-value does not tell you the probability that the null hypothesis is true. In other words, if you calculate a p-value of .04, this does not mean that the probability that the null hypothesis is true is 4%. Rather, it means that *if the null hypothesis was true*, the probability of obtaining the result you got is 4%. Now, this does indeed set off our bullshit detector, but again, it does not mean that the probability that the null hypothesis is true is 4%.

Let me convince you of this with a short example. Imagine that you and your partner have been trying to have a baby for the past year. One day, your partner calls you and says "Guess what! I took a pregnancy test and it came back positive!! I'm pregnant!!" So, given the positive pregnancy test, what is the probability that your partner is really pregnant?

Now, imagine that the pregnancy test your partner took gives incorrect results in 1% of cases. In other words, if you *are* pregnant, there is a 1% chance that the test will make a mistake and say that you are *not* pregnant. If you really are *not* pregnant, there is a 1% change that the test make a mistake and say you *are* pregnant.

Ok, so in this case, the null hypothesis here is that your partner is *not* pregnant, and the alternative hypothesis is that they *are* pregnant. Now, if the null hypothesis is true, then the probability that they would have gotten an (incorrect) positive test result is just 1%. Does this mean that the probability that your partner is *not* pregnant is only 1%.

No. Your partner is a man. The probability that the null hypothesis is true (i.e. that he is not pregnant), is 100%, not 1%. Your stupid boyfriend doesn't understand basic biology and decided to buy an expensive pregnancy test anyway.

This is an extreme example of course – in most tests that you do, there will be some positive probability that the null hypothesis is false. However, in order to reasonably calculate an accurate probability that the null hypothesis is true after collecting data, you *must* take into account the *prior* probability that the null hypothesis was true before you collected data. The method we use to do this is with Bayesian statistics. We'll go over Bayesian statistics in a later chapter.

13.2 Hypothesis test objects: `htest`

R stores hypothesis tests in special object classes called `htest`. `htest` objects contain all the major results from a hypothesis test, from the test statistic (e.g.; a t-statistic for a t-test, or a correlation coefficient for a correlation test), to the p-value, to a confidence interval. To show you how this works, let's create an `htest` object called `height.htest` containing the results from a two-sample t-test comparing the heights of male and female pirates:

```
# T-test comparing male and female heights
#  stored in a new htest object called height.htest
height.htest <- t.test(formula = height ~ sex,
                       data = pirates,
                       subset = sex %in% c("male", "female"))
```

Once you've created an `htest` object, you can view a print-out of the main results by just evaluating the object name:

```
# Print main results from height.htest
height.htest
##
##  Welch Two Sample t-test
##
## data:  height by sex
## t = -20, df = 1000, p-value <2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -15 -13
## sample estimates:
## mean in group female   mean in group male
##                  163                  177
```

Just like in dataframes, you can also access specific elements of the `htest` object by using the `$` symbol. To see all the named elements in the object, run `names()`:

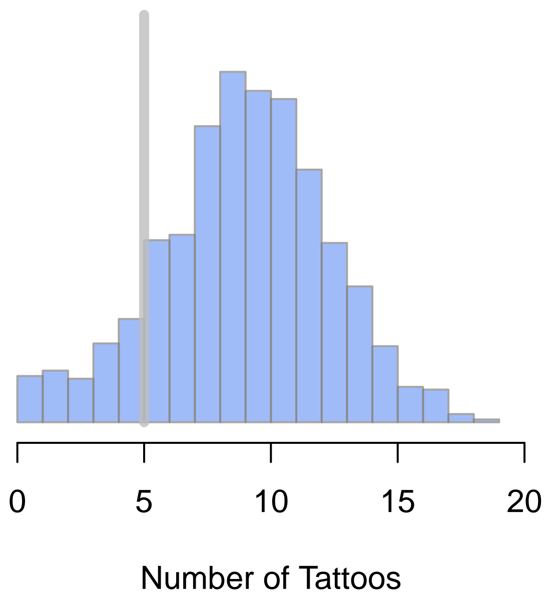
```
# Show me all the elements in the height.htest object
names(height.htest)
## [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
## [6] "null.value"  "alternative" "method"      "data.name"
```

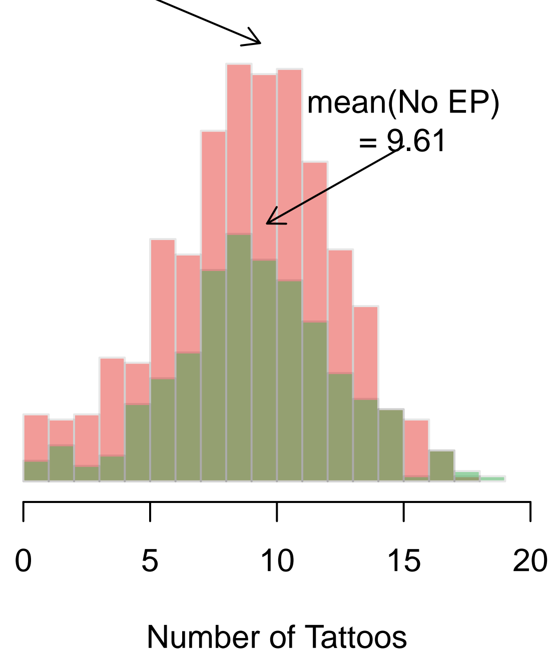
Now, if we want to access the test statistic or p-value directly, we can just use `$`:

```
# Get the test statistic
height.htest$statistic
##   t
## -21

# Get the p-value
height.htest$p.value
## [1] 1.4e-78

# Get a confidence interval for the mean
height.htest$conf.int
## [1] -15 -13
## attr(,"conf.level")
## [1] 0.95
```

13.3 T-test: `t.test()`

1–Sample t–test **2–Sample t–test**





To compare the mean of 1 group to a specific value, or to compare the means of 2 groups, you do a *t-test*. The t-test function in R is `t.test()`. The `t.test()` function can take several arguments, here I'll emphasize a few of them. To see them all, check the help menu for t.test (`?t.test`).

13.3.1 1-sample t-test

Argument	Description
`x`	A vector of data whose mean you want to compare to the null hypothesis `mu`
`mu`	The population mean under the null hypothesis. For example, `mu = 0` will test the null hypothesis that the true population mean is 0.
`alternative`	A string specifying the alternative hypothesis. Can be `"two.sided"` indicating a two-tailed test, or `"greater"` or “`less"` for a one-tailed test.

In a one-sample t-test, you compare the data from one group of data to some hypothesized mean. For example, if someone said that pirates on average have 5 tattoos, we could conduct a one-sample test comparing the data from a sample of pirates to a hypothesized mean of 5. To conduct a one-sample t-test in R using `t.test()`, enter a vector as the main argument `x`, and the null hypothesis as the argument `mu`

Here, I'll conduct a t-test to see if the average number of tattoos owned by pirates is different from 5:

```
tattoo.ttest <- t.test(x = pirates$tattoos,  # Vector of data
                       mu = 5)               # Null: Mean is 5

# Print the result
tattoo.ttest
##
##  One Sample t-test
##
## data:  pirates$tattoos
## t = 40, df = 1000, p-value <2e-16
## alternative hypothesis: true mean is not equal to 5
## 95 percent confidence interval:
##  9.2 9.6
## sample estimates:
## mean of x
##       9.4
```

As you can see, the function printed lots of information: the sample mean was 9.43, the test statistic (41.59), and the p-value was `2e-16` (which is virtually 0). Because 2e-16 is less than 0.05, we would reject the null hypothesis that the true mean is equal to 5.

Now, what happens if I change the null hypothesis to a mean of 9.4? Because the sample mean was 9.43, quite close to 9.4, the test statistic should decrease, and the p-value should increase:

```
tattoo.ttest <- t.test(x = pirates$tattoos,
                       mu = 9.5)  # Null: Mean is 9.4

tattoo.ttest
##
##  One Sample t-test
##
## data:  pirates$tattoos
## t = -0.7, df = 1000, p-value = 0.5
## alternative hypothesis: true mean is not equal to 9.5
## 95 percent confidence interval:
##  9.2 9.6
## sample estimates:
## mean of x
##       9.4
```

Just as we predicted! The test statistic decreased to just -0.67, and the p-value increased to 0.51. In other words, our sample mean of 9.43 is reasonably consistent with the hypothesis that the true population mean is 9.50.

13.3.2 2-sample t-test

In a two-sample t-test, you compare the means of two groups of data and test whether or not they are the same. We can specify two-sample t-tests in one of two ways. If the dependent and independent variables are in a dataframe, you can use the formula notation in the form `y ~ x`, and specify the dataset containing the data in `data`

```
# Fomulation of a two-sample t-test

# Method 1: Formula
t.test(formula = y ~ x,  # Formula
       data = df) # Dataframe containing the variables
```

Alternatively, if the data you want to compare are in individual vectors (not together in a dataframe), you can use the vector notation:

```
# Method 2: Vector
t.test(x = x,  # First vector
       y = y)  # Second vector
```

For example, let's test a prediction that pirates who wear eye patches have fewer tattoos on average than those who don't wear eye patches. Because the data are in the `pirates` dataframe, we can do this using the formula method:

```
# 2-sample t-test
#  IV = eyepatch (0 or 1)
#  DV = tattoos

tat.patch.htest <- t.test(formula = tattoos ~ eyepatch,
                          data = pirates)
```

This test gave us a test statistic of 1.22 and a p-value of 0.22. Because the p-value is greater than 0.05, we would fail to reject the null hypothesis.

```
# Show me all of the elements in the htest object
names(tat.patch.htest)
## [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
## [6] "null.value"  "alternative" "method"      "data.name"
```

Now, we can, for example, access the confidence interval for the mean differences using `$`

```
# Confidence interval for mean differences
tat.patch.htest$conf.int
## [1] -0.16  0.71
## attr(,"conf.level")
## [1] 0.95
```

13.3.2.1 Using `subset` to select levels of an IV

If your independent variable has more than two values, the `t.test()` function will return an error because it doesn't know which two groups you want to compare. For example, let's say I want to compare the

number of tattoos of pirates of different ages. Now, the age column has many different values, so if I don't tell `t.test()` which two values of `age` I want to compare, I will get an error like this:

```
# Will return an error because there are more than
#  2 levels of the age IV

t.test(formula = tattoos ~ age,
       data = pirates)
```

To fix this, I need to tell the `t.test()` function which two values of `age` I want to test. To do this, use the `subset` argument and indicate which values of the IV you want to test using the `%in%` operator. For example, to compare the number of tattoos between pirates of age 29 and 30, I would add the `subset = age %in% c(29, 30)` argument like this:

```
# Compare the tattoos of pirates aged 29 and 30:
tatage.htest <- t.test(formula = tattoos ~ age,
                        data = pirates,
                        subset = age %in% c(29, 30))  # Compare age of 29 to 30

tatage.htest
##
##  Welch Two Sample t-test
##
## data:  tattoos by age
## t = 0.3, df = 100, p-value = 0.8
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.1  1.4
## sample estimates:
## mean in group 29 mean in group 30
##             10.1              9.9
```

Looks like we got a p-value of 0.79 which is pretty high and suggests that we should fail to reject the null hypothesis.

You can select any subset of data in the `subset` argument to the `t.test()` function – not just the primary independent variable. For example, if I wanted to compare the number of tattoos between pirates who wear headbands or not, but only for female pirates, I would do the following

```
# Is there an effect of college on # of tattoos
#  only for female pirates?

t.test(formula = tattoos ~ college,
       data = pirates,
       subset = sex == "female")
##
##  Welch Two Sample t-test
##
## data:  tattoos by college
## t = 1, df = 500, p-value = 0.3
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.27  0.92
## sample estimates:
##  mean in group CCCC mean in group JSSFP
##                 9.6                 9.3
```

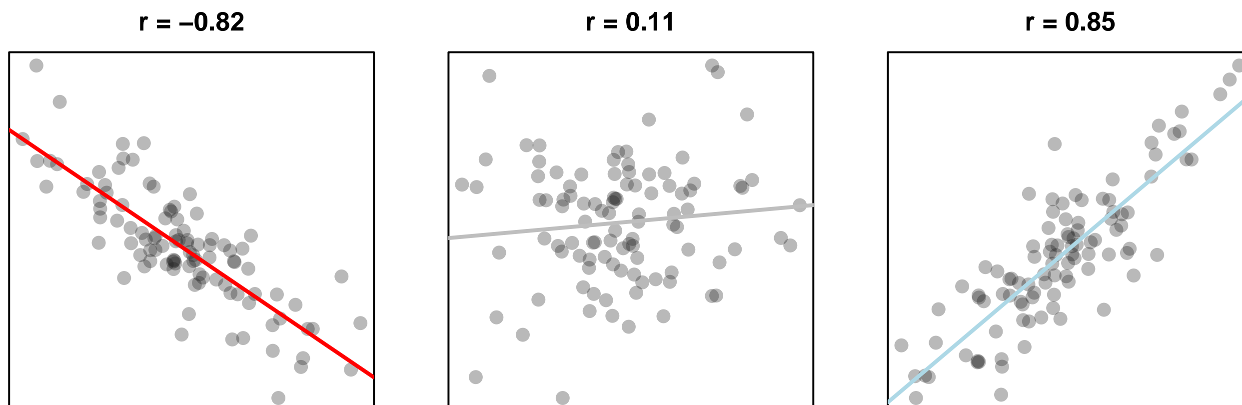


Figure 13.5: Three different correlations. A strong negative correlation, a very small positive correlation, and a strong positive correlation.

13.4 Correlation: `cor.test()`

Argument	Description
`formula`	A formula in the form `~ x + y`, where x and y are the names of the two variables you are testing. These variables should be two separate columns in a dataframe.
`data`	The dataframe containing the variables x and y
`alternative`	A string specifying the alternative hypothesis. Can be `"two.sided"` indicating a two-tailed test, or `"greater"` or “`less"` for a one-tailed test.
`method`	A string indicating which correlation coefficient to calculate and test. `"pearson"` (the default) stands for Pearson, while `"kendall"` and `"spearman"` stand for Kendall and Spearman correlations respectively.
`subset`	A vector specifying a subset of observations to use. E.g.; `subset = sex == "female"`

Next we'll cover two-sample correlation tests. In a correlation test, you are accessing the relationship between two variables on a ratio or interval scale: like height and weight, or income and beard length. The test statistic in a correlation test is called a *correlation coefficient* and is represented by the letter r. The coefficient can range from -1 to +1, with -1 meaning a strong negative relationship, and +1 meaning a strong positive relationship. The null hypothesis in a correlation test is a correlation of 0, which means no relationship at all:

To run a correlation test between two variables x and y, use the `cor.test()` function. You can do this in one of two ways, if x and y are columns in a dataframe, use the formula notation (`formula = ~ x + y`). If x and y are separate vectors (not in a dataframe), use the vector notation (`x, y`):

```
# Correlation Test
#   Correlating two variables x and y

# Method 1: Formula notation
##  Use if x and y are in a dataframe
cor.test(formula = ~ x + y,
         data = df)
```

```
# Method 2: Vector notation
## Use if x and y are separate vectors
cor.test(x = x,
         y = y)
```

Let's conduct a correlation test on the `pirates` dataset to see if there is a relationship between a pirate's age and number of parrots they've had in their lifetime. Because the variables (age and parrots) are in a dataframe, we can do this in formula notation:

```
# Is there a correlation between a pirate's age and
#  the number of parrots (s)he's owned?

# Method 1: Formula notation
age.parrots.ctest <- cor.test(formula = ~ age + parrots,
                              data = pirates)
# Print result
age.parrots.ctest
##
##  Pearson's product-moment correlation
##
## data:  age and parrots
## t = 6, df = 1000, p-value = 1e-09
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.13 0.25
## sample estimates:
##  cor
## 0.19
```

We can also do the same thing using vector notation – the results will be exactly the same:

```
# Method 2: Vector notation
age.parrots.ctest <- cor.test(x = pirates$age,
                              y = pirates$parrots)

# Print result
age.parrots.ctest
##
##  Pearson's product-moment correlation
##
## data:  pirates$age and pirates$parrots
## t = 6, df = 1000, p-value = 1e-09
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.13 0.25
## sample estimates:
##  cor
## 0.19
```

Looks like we have a positive correlation of 0.19 and a very small p-value. To see what information we can extract for this test, let's run the command `names()` on the test object:

```
names(age.parrots.ctest)
## [1] "statistic"   "parameter"   "p.value"     "estimate"    "null.value"
## [6] "alternative" "method"      "data.name"   "conf.int"
```

Looks like we've got a lot of information in this test object. As an example, let's look at the confidence interval for the population correlation coefficient:

```
#  95% confidence interval of the correlation
#   coefficient
age.parrots.ctest$conf.int
## [1] 0.13 0.25
## attr(,"conf.level")
## [1] 0.95
```

Just like the `t.test()` function, we can use the `subset` argument in the `cor.test()` function to conduct a test on a subset of the entire dataframe. For example, to run the same correlation test between a pirate's age and the number of parrot's she's owned, but *only* for female pirates, I can add the `subset = sex == "female"` argument:

```
# Is there a correlation between age and
#  number parrots ONLY for female pirates?

cor.test(formula = ~ age + parrots,
         data = pirates,
         subset = sex == "female")
##
##  Pearson's product-moment correlation
##
## data:  age and parrots
## t = 5, df = 500, p-value = 4e-06
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.12 0.30
## sample estimates:
##  cor
## 0.21
```

The results look pretty much identical. In other words, the strength of the relationship between a pirate's age and the number of parrot's they've owned is pretty much the same for female pirates and pirates in general.

13.5 Chi-square: `chsq.test()`

Next, we'll cover chi-square tests. In a chi-square test test, we test whether or not there is a difference in the rates of outcomes on a nominal scale (like sex, eye color, first name etc.). The test statistic of a chi-square text is χ^2 and can range from 0 to Infinity. The null-hypothesis of a chi-square test is that χ^2 = 0 which means no relationship.

A key difference between the `chisq.test()` and the other hypothesis tests we've covered is that `chisq.test()` requires a *table* created using the `table()` function as its main argument. You'll see how this works when we get to the examples.

13.5.0.1 1-sample Chi-square test

If you conduct a 1-sample chi-square test, you are testing if there is a difference in the number of members of each category in the vector. Or in other words, are all category memberships equally prevalent? Here's the general form of a one-sample chi-square test:

```
# General form of a one-sample chi-square test
chisq.test(x = table(x))
```

As you can see, the main argument to `chisq.test()` should be a *table* of values created using the `table()` function. For example, let's conduct a chi-square test to see if all pirate colleges are equally prevalent in the `pirates` data. We'll start by creating a table of the college data:

```
# Frequency table of pirate colleges
table(pirates$college)
##
##  CCCC JSSFP
##   658   342
```

Just by looking at the table, it looks like pirates are much more likely to come from Captain Chunk's Cannon Crew (CCCC) than Jack Sparrow's School of Fashion and Piratery (JSSFP). For this reason, we should expect a very large test statistic and a very small p-value. Let's test it using the `chisq.test()` function.

```
# Are all colleges equally prevelant?
college.cstest <- chisq.test(x = table(pirates$college))

college.cstest
##
##  Chi-squared test for given probabilities
##
## data:  table(pirates$college)
## X-squared = 100, df = 1, p-value <2e-16
```

Indeed, with a test statistic of 99.86 and a tiny p-value, we can safely reject the null hypothesis and conclude that certain college *are* more popular than others.

13.5.0.2 2-sample Chi-square test

If you want to see if the frequency of one nominal variable depends on a second nominal variable, you'd conduct a 2-sample chi-square test. For example, we might want to know if there is a relationship between the college a pirate went to, and whether or not he/she wears an eyepatch. We can get a contingency table of the data from the `pirates` dataframe as follows:

```
# Do pirates that wear eyepatches have come from different colleges
#  than those that do not wear eyepatches?

table(pirates$eyepatch,
      pirates$college)
##
##     CCCC JSSFP
##   0  225   117
##   1  433   225
```

To conduct a chi-square test on these data, we will enter table of the two data vectors:

```
# Is there a relationship between a pirate's
# college and whether or not they wear an eyepatch?

colpatch.cstest <- chisq.test(x = table(pirates$college,
                                        pirates$eyepatch))
```

```
colpatch.cstest
##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  table(pirates$college, pirates$eyepatch)
## X-squared = 0, df = 1, p-value = 1
```

It looks like we got a test statistic of $\chi^2 = 0$ and a p-value of 1. At the traditional p = .05 threshold for significance, we would conclude that we fail to reject the null hypothesis and state that we do not have enough information to determine if pirates from different colleges differ in how likely they are to wear eye patches.

13.5.1 Getting APA-style conclusions with the apa function

Most people think that R pirates are a completely unhinged, drunken bunch of pillaging buffoons. But nothing could be further from the truth! R pirates are a very organized and formal people who like their statistical output to follow strict rules. The most famous rules are those written by the American Pirate Association (APA). These rules specify exactly how an R pirate should report the results of the most common hypothesis tests to her fellow pirates.

For example, in reporting a t-test, APA style dictates that the result should be in the form t(df) = X, p = Y (Z-tailed), where df is the degrees of freedom of the text, X is the test statistic, Y is the p-value, and Z is the number of tails in the test. Now you can of course read these values directly from the test result, but if you want to save some time and get the APA style conclusion quickly, just use the apa function. Here's how it works:

Consider the following two-sample t-test on the pirates dataset that compares whether or not there is a significant age difference between pirates who wear headbands and those who do not:

```
test.result <- t.test(age ~ headband,
                      data = pirates)
test.result
##
##  Welch Two Sample t-test
##
## data:  age by headband
## t = 0.4, df = 100, p-value = 0.7
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.0  1.5
## sample estimates:
##  mean in group no mean in group yes
##               28                27
```

It looks like the test statistic is 0.35, degrees of freedom is 135.47, and the p-value is 0.73. Let's see how the apa function gets these values directly from the test object:

```
yarrr::apa(test.result)
## [1] "mean difference = -0.22, t(135.47) = 0.35, p = 0.73 (2-tailed)"
```

As you can see, the apa function got the values we wanted and reported them in proper APA style. The apa function will even automatically adapt the output for Chi-Square and correlation tests if you enter such a test object. Let's see how it works on a correlation test where we correlate a pirate's age with the number of parrots she has owned:

```
# Print an APA style conclusion of the correlation
#  between a pirate's age and # of parrots
age.parrots.ctest <- cor.test(formula = ~ age + parrots,
                              data = pirates)

# Pring result
age.parrots.ctest
##
##  Pearson's product-moment correlation
##
## data:  age and parrots
## t = 6, df = 1000, p-value = 1e-09
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.13 0.25
## sample estimates:
##  cor
## 0.19

# Print the apa style conclusion!
yarrr::apa(age.parrots.ctest)
## [1] "r = 0.19, t(998) = 6.13, p < 0.01 (2-tailed)"
```

The apa function has a few optional arguments that control things like the number of significant digits in the output, and the number of tails in the test. Run `?apa` to see all the options.

13.6 Test your R might!

The following questions are based on data from either the `movies` or the `pirates` dataset in the yarrr package. Make sure to load the package first to get access to the data!

1. Do male pirates have significantly longer beards than female pirates? Test this by conducting a t-test on the relevant data in the pirates dataset. (Hint: You'll have to select just the female and male pirates and remove the 'other' ones using `subset()`)
2. Are pirates whose favorite pixar movie is Up more or less likely to wear an eye patch than those whose favorite pixar movie is Inside Out? Test this by conducting a chi-square test on the relevant data in the pirates dataset. (Hint: Create a new dataframe that only contains data from pirates whose favorite move is either Up or Inside Out using `subset()`. Then do the test on this new dataframe.)
3. Do longer movies have significantly higher budgets than shorter movies? Answer this question by conducting a correlation test on the appropriate data in the movies dataset.
4. Do R rated movies earn significantly more money than PG-13 movies? Test this by conducting a t-test on the relevant data in the movies dataset.
5. Are certain movie genres significantly more common than others in the movies dataset? Test this by conducting a 1-sample chi-square test on the relevant data in the movies dataset.
6. Do sequels and non-sequels differ in their ratings? Test this by conducting a 2-sample chi-square test on the relevant data in the movies dataset.

Chapter 14

ANOVA

In the last chapter we covered 1 and two sample hypothesis tests. In these tests, you are either comparing 1 group to a hypothesized value, or comparing the relationship between two groups (either their means or their correlation). In this chapter, we'll cover how to analyse more complex experimental designs with ANOVAs.

When do you conduct an ANOVA? You conduct an ANOVA when you are testing the effect of one or more nominal (aka factor) independent variable(s) on a numerical dependent variable. A nominal (factor) variable is one that contains a finite number of categories with no inherent order. Gender, profession, experimental conditions, and Justin Bieber albums are good examples of factors (not necessarily of good music). If you only include one independent variable, this is called a *One-way ANOVA*. If you include two independent variables, this is called a *Two-way ANOVA*. If you include three independent variables it is called a *Menage a trois 'NOVA*.

Ok maybe it's not yet, but we repeat it enough it will be and we can change the world.

For example, let's say you want to test how well each of three different cleaning fluids are at getting poop off of your poop deck.To test this, you could do the following: over the course of 300 cleaning days, you clean different areas of the deck with the three different cleaners. You then record how long it takes for each cleaner to clean its portion of the deck. At the same time, you could also measure how well the cleaner is cleaning two different types of poop that typically show up on your deck: shark and parrot. Here, your independent variables *cleaner* and *type* are factors, and your dependent variable *time* is numeric.

Thankfully, this experiment has already been conducted. The data are recorded in a dataframe called `poopdeck` in the yarrr package. Here's how the first few rows of the data look:

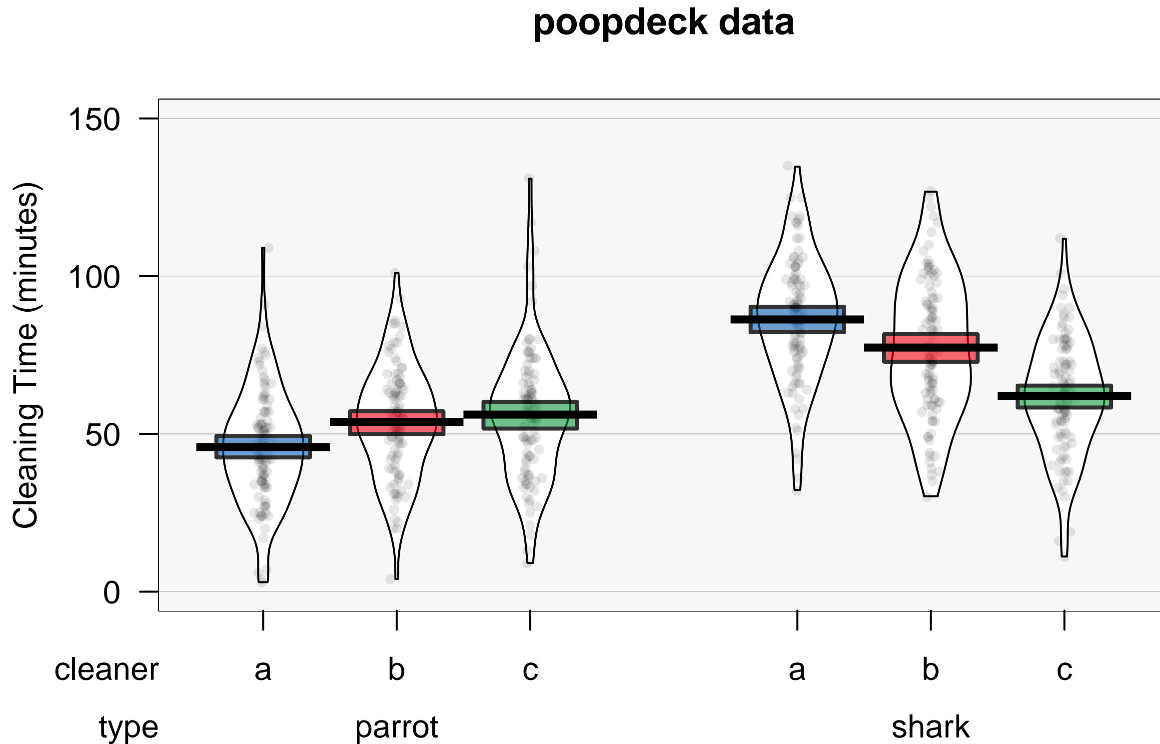
```
head(poopdeck)
##   day cleaner   type time int.fit me.fit
## 1   1       a parrot   47      46     54
## 2   1       b parrot   55      54     54
## 3   1       c parrot   64      56     47
## 4   1       a  shark  101      86     78
## 5   1       b  shark   76      77     77
## 6   1       c  shark   63      62     71
```

We can visualize the poopdeck data using (of course) a pirate plot:

```
pirateplot(formula = time ~ cleaner + type,
           data = poopdeck,
           ylim = c(0, 150),
           xlab = "Cleaner",
           ylab = "Cleaning Time (minutes)",
```

Figure 14.1: Menage a trois wine – the perfect pairing for a 3-way ANOVA

```
         main = "poopdeck data",
         back.col = gray(.97),
         cap.beans = TRUE,
         theme = 2)
```



Given this data, we can use ANOVAs to answer four separate questions:

Question	Analysis	Formula
Is there a difference between the different cleaners on cleaning time (ignoring poop type)?	One way ANOVA	`time ~ cleaner`
Is there a difference between the different poop types on cleaning time (ignoring which cleaner is used)	One-way ANOVA	`time ~ type`
Is there a *unique* effect of the cleaner or poop types on cleaning time?	Two-way ANOVA	`time ~ cleaner + type`
Does the effect of cleaner depend on the poop type?	Two-way ANOVA with interaction term	`time ~ cleaner * type`

14.1 Full-factorial between-subjects ANOVA

There are many types of ANOVAs that depend on the type of data you are analyzing. In fact, there are so many types of ANOVAs that there are entire books explaining differences between one type and another. For this book, we'll cover just one type of ANOVAs called *full-factorial, between-subjects ANOVAs*. These are the simplest types of ANOVAs which are used to analyze a standard experimental design. In a full-factorial, between-subjects ANOVA, participants (aka, source of data) are randomly assigned to a unique combination of factors – where a combination of factors means a specific experimental condition.

For example, consider a psychology study comparing the effects of caffeine on cognitive performance. The study could have two independent variables: drink type (soda vs. coffee vs. energy drink), and drink dose (.25l, .5l, 1l). In a full-factorial design, each participant in the study would be randomly assigned to one drink type and one drink dose condition. In this design, there would be 3 x 3 = 9 conditions.

For the rest of this chapter, I will refer to full-factorial between-subjects ANOVAs as 'standard' ANOVAs

14.1.1 What does ANOVA stand for?

ANOVA stands for "Analysis of variance." At first glance, this sounds like a strange name to give to a test that you use to find differences in **means**, not differences in **variances**. However, ANOVA actually uses variances to determine whether or not there are 'real' differences in the means of groups. Specifically, it looks at how variable data are *within* groups and compares that to the variability of data *between* groups. If the between-group variance is large compared to the within group variance, the ANOVA will conclude that the groups *do* differ in their means. If the between-group variance is small compared to the within group variance, the ANOVA will conclude that the groups are all the same. See Figure~**??** for a visual depiction of an ANOVA.

14.2 4 Steps to conduct an ANOVA

Here are the 4 steps you should follow to conduct a standard ANOVA in R:

1. Create an ANOVA object using the `aov()` function. In the `aov()` function, specify the independent and dependent variable(s) with a formula with the format `y ~ x1 + x2` where y is the dependent variable, and x1, x2 … are one (more more) factor independent variables.

```
# Step 1: Create an aov object
mod.aov <- aov(formula = y ~ x1 + x2 + ...,
               data = data)
```

2. Create a summary ANOVA table by applying the `summary()` function to the ANOVA object you created in Step 1.

```
# Step 2: Look at a summary of the aov object
summary(mod.aov)
```

3. If necessary, calculate post-hoc tests by applying a post-hoc testing function like `TukeyHSD()` to the ANOVA object you created in Step 1.

```
# Step 3: Calculate post-hoc tests
TukeyHSD(mod.aov)
```

4. If necessary, interpret the nature of the group differences by creating a linear regression object using `lm()` using the same arguments you used in the `aov()` function in Step 1.

```
# Step 4: Look at coefficients
mod.lm <- lm(formula = y ~ x1 + x2 + ...,
             data = data)

summary(mod.lm)
```

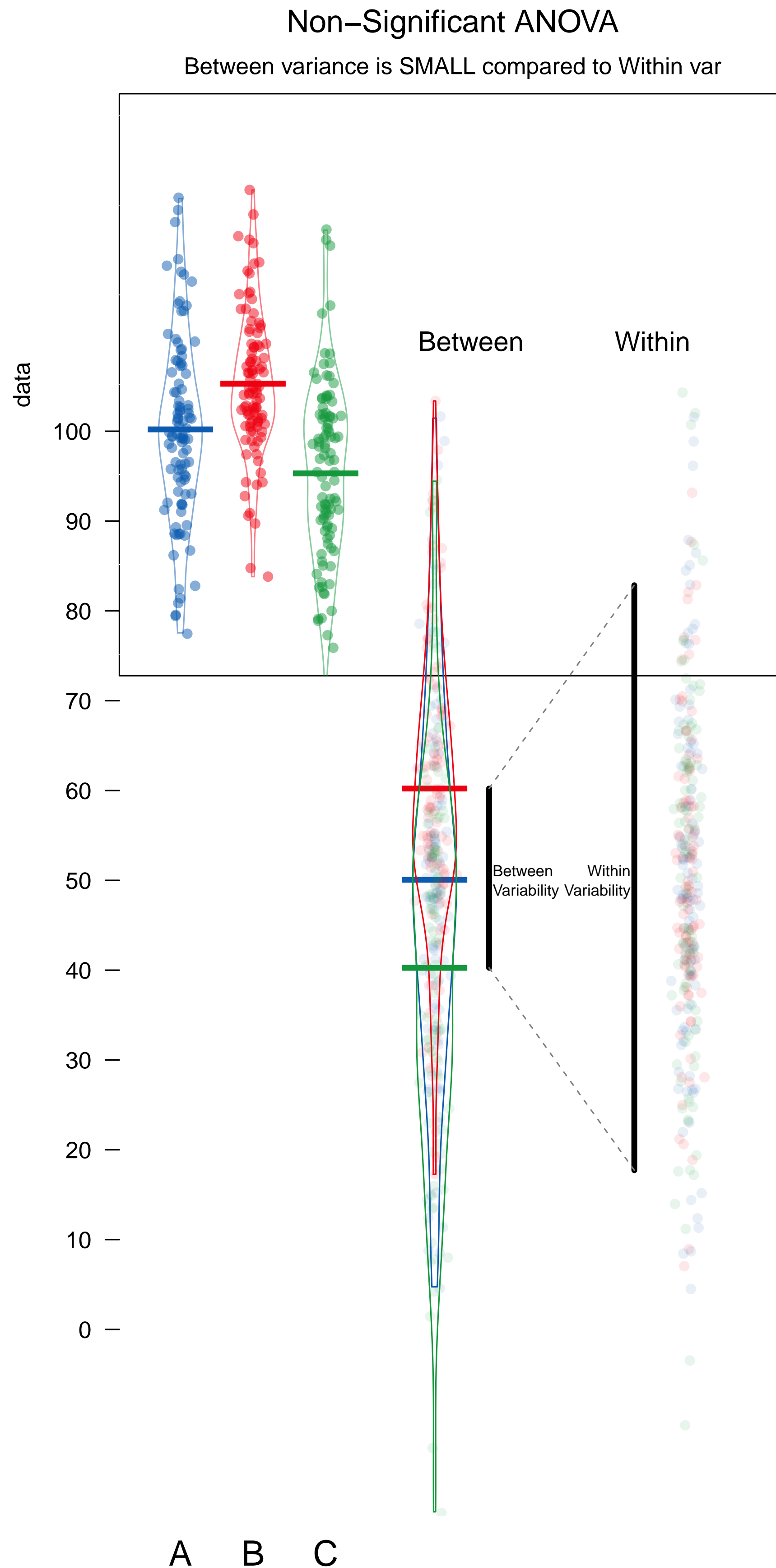


Figure 14.2: How ANOVAs work. ANOVA compares the variability between groups (i.e.; the differences in the group means) to the variability within groups (i.e.; how much individuals generally differ from each other). If the variability between groups is small compared to the variability between groups, ANOVA will return a non-significant result – suggesting that the groups are not really different. If the variability between groups is large compared to the variability within groups, ANOVA will return a significant result – indicating

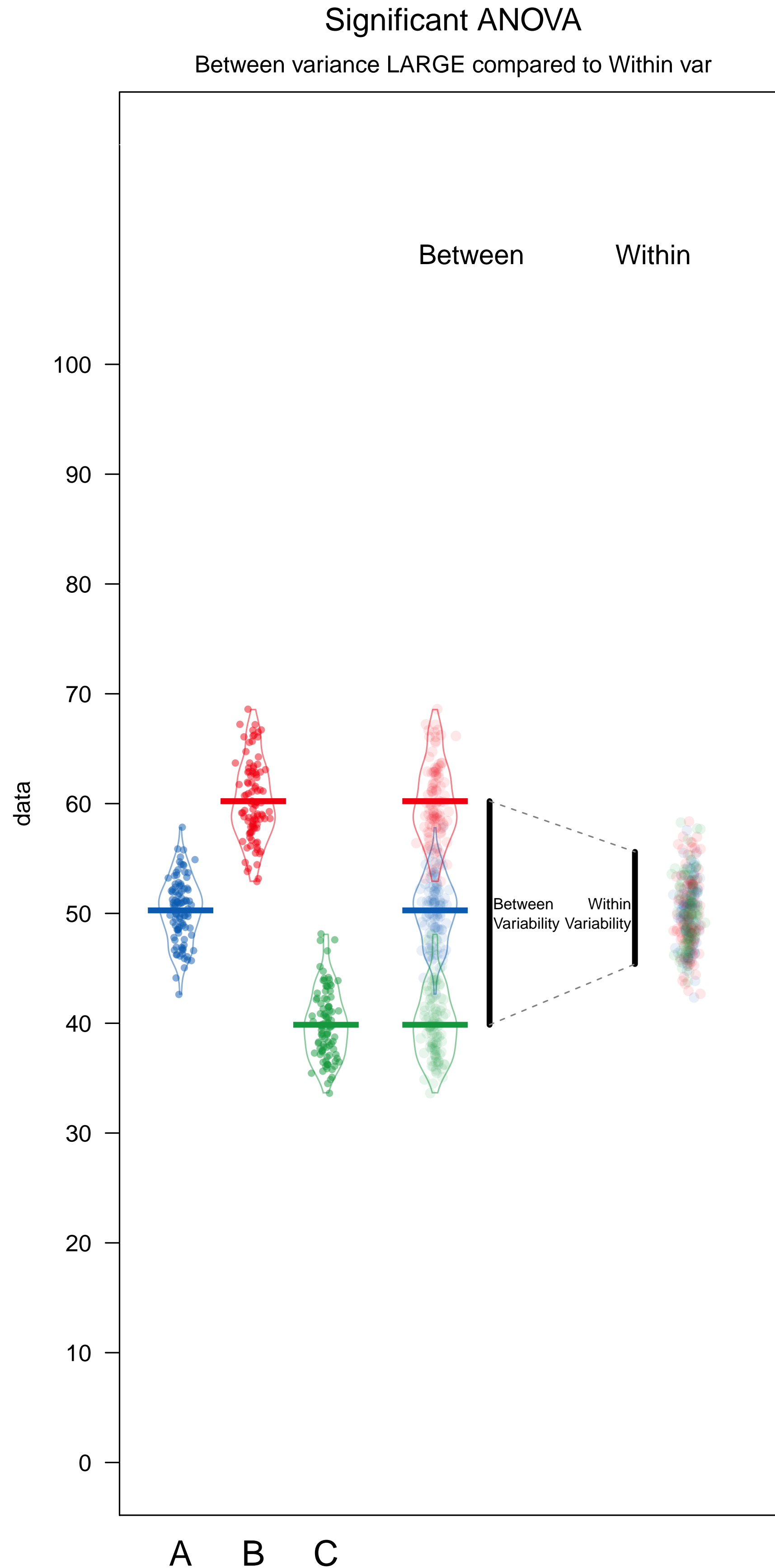
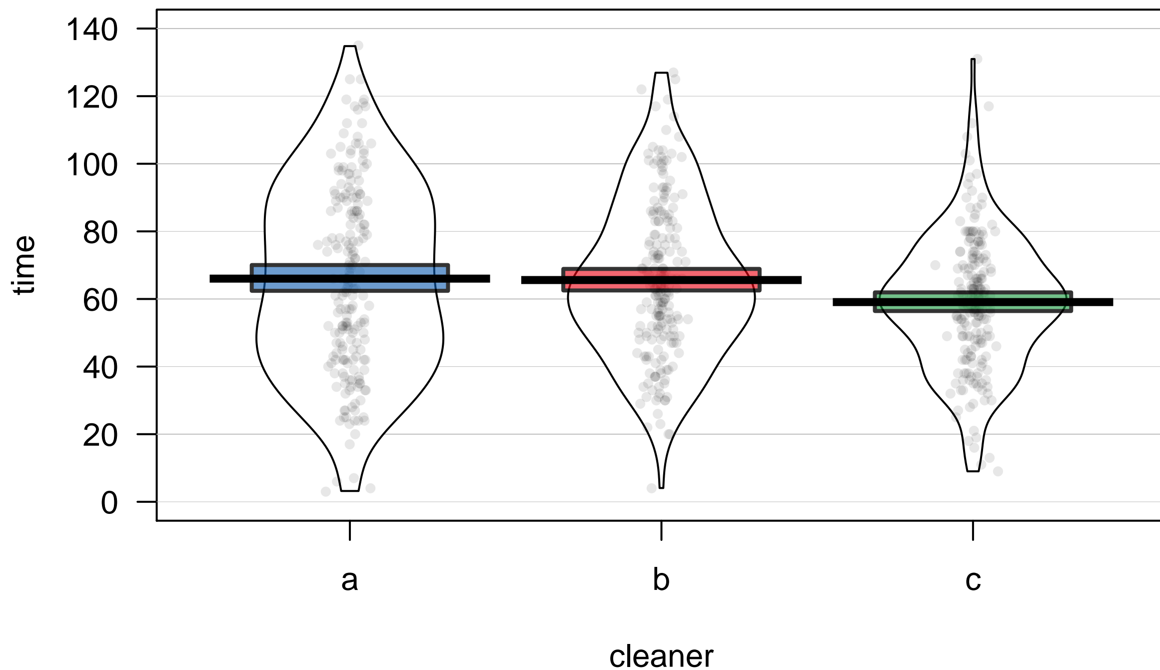


Figure 14.3: How ANOVAs work. ANOVA compares the variability between groups (i.e.; the differences in the group means) to the variability within groups (i.e.; how much individuals generally differ from each other). If the variability between groups is small compared to the variability between groups, ANOVA will return a non-significant result – suggesting that the groups are not really different. If the variability between groups is large compared to the variability within groups, ANOVA will return a significant result – indicating

14.3 Ex: One-way ANOVA

Let's do an example by running both a one-way ANOVA on the `poopdeck` data. We'll set cleaning time `time` as the dependent variable and the cleaner type `cleaner` as the independent variable. We can represent the data as a pirateplot:

```
yarrr::pirateplot(time ~ cleaner,
                  data = poopdeck,
                  theme = 2,
                  cap.beans = TRUE,
                  main = "formula = time ~ cleaner")
```



From the plot, it looks like cleaners a and b are the same, and cleaner c is a bit faster. To test this, we'll create an ANOVA object with `aov`. Because `time` is the dependent variable and `cleaner` is the independent variable, we'll set the formula to `formula = time ~ cleaner`

```
# Step 1: aov object with time as DV and cleaner as IV
cleaner.aov <- aov(formula = time ~ cleaner,
                   data = poopdeck)
```

Now, to see a full ANOVA summary table of the ANOVA object, apply the `summary()` to the ANOVA object from Step 1.

```
# Step 2: Look at the summary of the anova object
summary(cleaner.aov)
##              Df Sum Sq Mean Sq F value Pr(>F)
## cleaner       2   6057    3028    5.29 0.0053 **
## Residuals   597 341511     572
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The main result from our table is that we have a significant effect of cleaner on cleaning time (F(2, 597) =

5.29, p = 0.005. However, the ANOVA table does not tell us which levels of the independent variable differ. In other words, we don't know which cleaner is better than which. To answer this, we need to conduct a post-hoc test.

If you've found a significant effect of a factor, you can then do post-hoc tests to test the difference between each all pairs of levels of the independent variable. There are many types of pairwise comparisons that make different assumptions. To learn more about the logic behind different post-hoc tests, check out the Wikipedia page here: https://en.wikipedia.org/wiki/Post_hoc_analysis. One of the most common post-hoc tests for standard ANOVAs is the Tukey Honestly Significant Difference (HSD) test. To see additional information about the Tukey HSD test, check out the Wikipedia page here: https://en.wikipedia.org/wiki/Tukey's_range_test To do an HSD test, apply the `TukeyHSD()` function to your ANOVA object as follows:

```
# Step 3: Conduct post-hoc tests
TukeyHSD(cleaner.aov)
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = time ~ cleaner, data = poopdeck)
##
## $cleaner
##      diff lwr  upr p adj
## b-a -0.42  -6  5.2  0.98
## c-a -6.94 -13 -1.3  0.01
## c-b -6.52 -12 -0.9  0.02
```

This table shows us pair-wise differences between each group pair. The `diff` column shows us the mean differences between groups (which thankfully are identical to what we found in the summary of the regression object before), a confidence interval for the difference, and a p-value testing the null hypothesis that the group differences are not different.

I almost always find it helpful to combine an ANOVA summary table with a regression summary table. Because ANOVA is just a special case of regression (where all the independent variables are factors), you'll get the same results with a regression object as you will with an ANOVA object. However, the format of the results are different and frequently easier to interpret.

To create a regression object, use the `lm()` function. Your inputs to this function will be *identical* to your inputs to the `aov()` function

```
# Step 4: Create a regression object
cleaner.lm <- lm(formula = time ~ cleaner,
                 data = poopdeck)

# Show summary
summary(cleaner.lm)
##
## Call:
## lm(formula = time ~ cleaner, data = poopdeck)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -63.02 -16.60  -1.05  16.92  71.92
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    66.02       1.69   39.04   <2e-16 ***
## cleanerb       -0.42       2.39   -0.18   0.8607
```

```
## cleanerc       -6.94       2.39   -2.90   0.0038 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 24 on 597 degrees of freedom
## Multiple R-squared:  0.0174, Adjusted R-squared:  0.0141
## F-statistic: 5.29 on 2 and 597 DF,  p-value: 0.00526
```

As you can see, the regression table does not give us tests for each variable like the ANOVA table does. Instead, it tells us how different each level of an independent variable is from a *default* value. You can tell which value of an independent variable is the default variable just by seeing which value is missing from the table. In this case, I don't see a coefficient for cleaner a, so that must be the default value.

The intercept in the table tells us the mean of the default value. In this case, the mean time of cleaner a was 66.02. The coefficients for the other levels tell us that cleaner b is, on average 0.42 minutes faster than cleaner a, and cleaner c is on average 6.94 minutes faster than cleaner a. Not surprisingly, these are the same differences we saw in the Tukey HSD test!

14.4 Ex: Two-way ANOVA

To conduct a two-way ANOVA or a Menage a trois NOVA, just include additional independent variables in the regression model formula with the + sign. That's it. All the steps are the same. Let's conduct a two-way ANOVA with both cleaner and type as independent variables. To do this, we'll set `formula = time ~ cleaner + type`.

```
# Step 1: Create ANOVA object with aov()
cleaner.type.aov <- aov(formula = time ~ cleaner + type,
                        data = poopdeck)
```

```
# Step 2: Get ANOVA table with summary()
summary(cleaner.type.aov)
##              Df Sum Sq Mean Sq F value Pr(>F)
## cleaner       2   6057    3028    6.94  0.001 **
## type          1  81620   81620  187.18 <2e-16 ***
## Residuals   596 259891     436
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

It looks like we found significant effects of both independent variables.

```
# Step 3: Conduct post-hoc tests
TukeyHSD(cleaner.type.aov)
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = time ~ cleaner + type, data = poopdeck)
##
## $cleaner
##      diff   lwr  upr p adj
## b-a -0.42  -5.3  4.5  0.98
## c-a -6.94 -11.8 -2.0  0.00
## c-b -6.52 -11.4 -1.6  0.01
##
## $type
```

```
##               diff lwr upr p adj
## shark-parrot   23  20  27     0
```

The only non-significant group difference we found is between cleaner b and cleaner a. All other comparisons were significant.

```
# Step 4: Look at regression coefficients
cleaner.type.lm <- lm(formula = time ~ cleaner + type,
                      data = poopdeck)

summary(cleaner.type.lm)
##
## Call:
## lm(formula = time ~ cleaner + type, data = poopdeck)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -59.74 -13.79  -0.68  13.58  83.58
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    54.36       1.71   31.88  < 2e-16 ***
## cleanerb       -0.42       2.09   -0.20  0.84067
## cleanerc       -6.94       2.09   -3.32  0.00094 ***
## typeshark      23.33       1.71   13.68  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 21 on 596 degrees of freedom
## Multiple R-squared:  0.252,  Adjusted R-squared:  0.248
## F-statistic:   67 on 3 and 596 DF,  p-value: <2e-16
```

Now we need to interpret the results in respect to two default values (here, cleaner = a and type = parrot). The intercept means that the average time for cleaner a on parrot poop was 54.36 minutes. Additionally, the average time to clean shark poop was 23.33 minutes slower than when cleaning parrot poop.

14.4.1 ANOVA with interactions

Interactions between variables test whether or not the effect of one variable depends on another variable. For example, we could use an interaction to answer the question: *Does the effect of cleaners depend on the type of poop they are used to clean?* To include interaction terms in an ANOVA, just use an asterix (*) instead of the plus (+) between the terms in your formula. Note that when you include an interaction term in a regression object, R will automatically include the main effects as well/

Let's repeat our previous ANOVA with two independent variables, but now we'll include the interaction between cleaner and type. To do this, we'll set the formula to `time ~ cleaner * type`.

```
# Step 1: Create ANOVA object with interactions
cleaner.type.int.aov <- aov(formula = time ~ cleaner * type,
                            data = poopdeck)

# Step 2: Look at summary table
summary(cleaner.type.int.aov)
##                Df Sum Sq Mean Sq F value  Pr(>F)
## cleaner         2   6057    3028    7.82 0.00044 ***
```

```
## type           1  81620   81620  210.86 < 2e-16 ***
## cleaner:type   2  29968   14984   38.71 < 2e-16 ***
## Residuals    594 229923     387
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Looks like we did indeed find a significant interaction between cleaner and type. In other words, the effectiveness of a cleaner depends on the type of poop it's being applied to. This makes sense given our plot of the data at the beginning of the chapter.

To understand the nature of the difference, we'll look at the regression coefficients from a regression object:

```
# Step 4: Calculate regression coefficients
cleaner.type.int.lm <- lm(formula = time ~ cleaner * type,
                          data = poopdeck)

summary(cleaner.type.int.lm)
##
## Call:
## lm(formula = time ~ cleaner * type, data = poopdeck)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -54.28 -12.83  -0.08  12.29  74.87
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)           45.76       1.97   23.26  < 2e-16 ***
## cleanerb               8.06       2.78    2.90  0.00391 **
## cleanerc              10.37       2.78    3.73  0.00021 ***
## typeshark             40.52       2.78   14.56  < 2e-16 ***
## cleanerb:typeshark   -16.96       3.93   -4.31  1.9e-05 ***
## cleanerc:typeshark   -34.62       3.93   -8.80  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 20 on 594 degrees of freedom
## Multiple R-squared:  0.338,  Adjusted R-squared:  0.333
## F-statistic: 60.8 on 5 and 594 DF,  p-value: <2e-16
```

Again, to interpret this table, we first need to know what the default values are. We can tell this from the coefficients that are 'missing' from the table. Because I don't see terms for `cleanera` or `typeparrot`, this means that `cleaner = "a"` and `type = "parrot"` are the defaults. Again, we can interpret the coefficients as `differences` between a level and the default. It looks like for parrot poop, cleaners b and c both take more time than cleaner a (the default). Additionally, shark poop tends to take much longer than parrot poop to clean (the estimate for typeshark is positive).

The interaction terms tell us how the effect of cleaners `changes` when one is cleaning shark poop. The negative estimate (-16.96) for `cleanerb:typeshark` means that cleaner b is, on average 16.96 minutes `faster` when cleaning shark poop compared to parrot poop. Because the previous estimate for cleaner b (for parrot poop) was just 8.06, this suggests that cleaner b is `slower` than cleaner a for parrot poop, but `faster` than cleaner a for shark poop. Same thing for cleaner c which simply has stronger effects in both directions.

14.5 Type I, Type II, and Type III ANOVAs

It turns out that there is not just one way to calculate ANOVAs. In fact, there are three different types - called, Type 1, 2, and 3 (or Type I, II and III). These types differ in how they calculate variability (specifically the `sums of of squares`). If your data is relatively `balanced`, meaning that there are relatively equal numbers of observations in each group, then all three types will give you the same answer. However, if your data are `unbalanced`, meaning that some groups of data have many more observations than others, then you need to use Type II (2) or Type III (3).

The standard `aov()` function in base-R uses Type I sums of squares. Therefore, it is only appropriate when your data are balanced. If your data are unbalanced, you should conduct an ANOVA with Type II or Type III sums of squares. To do this, you can use the `Anova()` function in the `car` package. The `Anova()` function has an argument called `type` that allows you to specify the type of ANOVA you want to calculate.

In the next code chunk, I'll calculate 3 separate ANOVAs from the poopdeck data using the three different types. First, I'll create a regression object with `lm()`. As you'll see, the `Anova()` function requires you to enter a regression object as the main argument, and `not` a formula and dataset. That is, you need to first create a regression object from the data with `lm()` (or `glm()`), and then enter that object into the `Anova()` function. You can also do the same thing with the standard `aov()` function'.

```
# Step 1: Calculate regression object with lm()
time.lm <- lm(formula = time ~ type + cleaner,
              data = poopdeck)
```

Now that I've created the regression object `time.lm`, I can calculate the three different types of ANOVAs by entering the object as the main argument to either `aov()` for a Type I ANOVA, or `Anova()` in the car package for a Type II or Type III ANOVA:

```
# Type I ANOVA - aov()
time.I.aov <- aov(time.lm)

# Type II ANOVA - Anova(type = 2)
time.II.aov <- car::Anova(time.lm, type = 2)

# Type III ANOVA - Anova(type = 3)
time.III.aov <- car::Anova(time.lm, type = 3)
```

As it happens, the data in the poopdeck dataframe are perfectly balanced (so we'll get exactly the same result for each ANOVA type. However, if they were not balanced, then we should *not* use the Type I ANOVA calculated with the *aov()* function.

To see if your data are balanced, you can use the `table()` function:

```
# Are observations in the poopdeck data balanced?
with(poopdeck,
     table(cleaner, type))
##        type
## cleaner parrot shark
##       a    100   100
##       b    100   100
##       c    100   100
```

As you can see, in the poopdeck data, the observations are perfectly balanced, so it doesn't matter which type of ANOVA we use to analyse the data.

For more detail on the different types, check out https://mcfromnz.wordpress.com/2011/03/02/anova-type-iiiiii-ss-explained/.

14.6 Getting additional information from ANOVA objects

You can get a lot of interesting information from ANOVA objects. To see everything that's stored in one, run the `names()` command on an ANOVA object. For example, here's what's in our last ANOVA object:

```
# Show me what's in my aov object
names(cleaner.type.int.aov)
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "contrasts"     "xlevels"       "call"          "terms"
## [13] "model"
```

For example, the `"fitted.values"` contains the model fits for the dependent variable (time) for every observation in our dataset. We can add these fits back to the dataset with the $ operator and assignment. For example, let's get the model fitted values from both the interaction model (cleaner.type.aov) and the non-interaction model (cleaner.type.int.aov) and assign them to new columns in the dataframe:

```
# Add the fits for the interaction model to the dataframe as int.fit

poopdeck$int.fit <- cleaner.type.int.aov$fitted.values

# Add the fits for the main effects model to the dataframe as me.fit

poopdeck$me.fit <- cleaner.type.aov$fitted.values
```

Now let's look at the first few rows in the table to see the fits for the first few observations.

```
head(poopdeck)
##   day cleaner   type time int.fit me.fit
## 1   1       a parrot   47      46     54
## 2   1       b parrot   55      54     54
## 3   1       c parrot   64      56     47
## 4   1       a  shark  101      86     78
## 5   1       b  shark   76      77     77
## 6   1       c  shark   63      62     71
```

You can use these fits to see how well (or poorly) the model(s) were able to fit the data. For example, we can calculate how far each model's fits were from the true data as follows:

```
# How far were the interaction model fits from the data on average?

mean(abs(poopdeck$int.fit - poopdeck$time))
## [1] 15

# How far were the main effect model fits from the data on average?

mean(abs(poopdeck$me.fit - poopdeck$time))
## [1] 17
```

As you can see, the interaction model was off from the data by 15.35 minutes on average, while the main effects model was off from the data by 16.54 on average. This is not surprising as the interaction model is more complex than the main effects only model. However, just because the interaction model is better at fitting the data doesn't necessarily mean that the interaction is either meaningful or reliable.

14.7 Repeated measures ANOVA using the lme4 package

If you are conducting an analyses where you're repeating measurements over one or more third variables, like giving the same participant different tests, you should do a mixed-effects regression analysis. To do this, you should use the `lmer` function in the `lme4` package. For example, in our `poopdeck` data, we have repeated measurements for days. That is, on each day, we had 6 measurements. Now, it's possible that the overall cleaning times differed depending on the day. We can account for this by including random intercepts for day by adding the `(1|day)` term to the formula specification. For more tips on mixed-effects analyses, check out this great tutorial by Bodo Winter at http://www.bodowinter.com/tutorial/bw_LME_tutorial2.pdf.

```
# install.packages(lme4)  # If you don't have the package already
library(lme4)

# Calculate a mixed-effects regression on time with
#  Two fixed factors (cleaner and type)
#  And one repeated measure (day)

my.mod <- lmer(formula = time ~ cleaner + type + (1|day),
               data = poopdeck)
```

14.8 Test your R might!

For the following questions, use the `pirates` dataframe in the `yarrr` package

1. Is there a significant relationship between a pirate's favorite pixar movie and the number of tattoos (s)he has? Conduct an appropriate ANOVA with `fav.pixar` as the independent variable, and `tattoos` as the dependent variable. If there is a significant relationship, conduct a post-hoc test to determine which levels of the independent variable(s) differ.

2. Is there a significant relationship between a pirate's favorite pirate and how many tattoos (s)he has? Conduct an appropriate ANOVA with `favorite.pirate` as the independent variable, and `tattoos` as the dependent variable. If there is a significant relationship, conduct a post-hoc test to determine which levels of the independent variable(s) differ.

3. Now, repeat your analysis from the previous two questions, but include both independent variables `fav.pixar` and `favorite.pirate` in the ANOVA. Do your conclusions differ when you include both variables?

4. Finally, test if there is an interaction between `fav.pixar` and `favorite.pirate` on number of tattoos.

Chapter 15

Regression

Pirates like diamonds. Who doesn't?! But as much as pirates love diamonds, they hate getting ripped off. For this reason, a pirate needs to know how to accurately assess the value of a diamond. For example, how much should a pirate pay for a diamond with a weight of 2.0 grams, a clarity value of 1.0, and a color gradient of 4 out of 10? To answer this, we'd like to know how the attributes of diamonds (e.g.; weight, clarity, color) relate to its value. We can get these values using linear regression.

15.1 The Linear Model

The linear model is easily the most famous and widely used model in all of statistics. Why? Because it can apply to so many interesting research questions where you are trying to predict a continuous variable of interest (the *response* or *dependent variable*) on the basis of one or more other variables (the *predictor* or *independent variables*).

The linear model takes the following form, where the x values represent the predictors, while the beta values represent weights.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... \beta_n x_n$$

For example, we could use a regression model to understand how the value of a diamond relates to two independent variables: its weight and clarity. In the model, we could define the value of a diamond as $\beta_{weight} \times weight + \beta_{clarity} \times clarity$. Where β_{weight} indicates how much a diamond's value changes as a function of its weight, and $\beta_{clarity}$ defines how much a diamond's value change as a function of its clarity.

15.2 Linear regression with `lm()`

Argument	Description
`formula`	A formula in the form `y ~ x1 + x2 + ...` where y is the dependent variable, and x1, x2, … are the independent variables. If you want to include all columns (excluding y) as independent variables, just enter `y ~ .`
`data`	The dataframe containing the columns specified in the formula.

To estimate the beta weights of a linear model in R, we use the `lm()` function. The function has three key arguments: `formula`, and `data`

Figure 15.1: Insert funny caption here.

15.2.1 Estimating the value of diamonds with `lm()`

We'll start with a simple example using a dataset in the yarrr package called `diamonds`. The dataset includes data on 150 diamonds sold at an auction. Here are the first few rows of the dataset:

```
library(yarrr)
head(diamonds)
##   weight clarity color value value.lm weight.c clarity.c value.g190
## 1    9.3    0.88     4   182      186    -0.55     -0.12      FALSE
## 2   11.1    1.05     5   191      193     1.20      0.05       TRUE
## 3    8.7    0.85     6   176      183    -1.25     -0.15      FALSE
## 4   10.4    1.15     5   195      194     0.53      0.15       TRUE
## 5   10.6    0.92     5   182      189     0.72     -0.08      FALSE
## 6   12.3    0.44     4   183      183     2.45     -0.56      FALSE
##   pred.g190
## 1     0.163
## 2     0.821
## 3     0.030
## 4     0.846
## 5     0.445
## 6     0.087
```

Our goal is to come up with a linear model we can use to estimate the value of each diamond (DV = `value`) as a linear combination of three independent variables: its weight, clarity, and color. The linear model will estimate each diamond's value using the following equation:

$$\beta_{Int} + \beta_{weight} \times weight + \beta_{clarity} \times clarity + \beta_{color} \times color$$

where β_{weight} is the increase in value for each increase of 1 in weight, $\beta_{clarity}$ is the increase in value for each increase of 1 in clarity (etc.). Finally, β_{Int} is the baseline value of a diamond with a value of 0 in all independent variables.

To estimate each of the 4 weights, we'll use `lm()`. Because `value` is the dependent variable, we'll specify the formula as `formula = value ~ weight + clarity + color`. We'll assign the result of the function to a new object called `diamonds.lm`:

```
# Create a linear model of diamond values
#   DV = value, IVs = weight, clarity, color

diamonds.lm <- lm(formula = value ~ weight + clarity + color,
                  data = diamonds)
```

To see the results of the regression analysis, including estimates for each of the beta values, we'll use the `summary()` function:

```
# Print summary statistics from diamond model
summary(diamonds.lm)
##
## Call:
## lm(formula = value ~ weight + clarity + color, data = diamonds)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -10.405  -3.547  -0.113   3.255  11.046
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  148.335      3.625   40.92   <2e-16 ***
```

Linear Model of Diamond Values

$$148.3 + 2.19 \times x_{weight} + 21.69 \times x_{clarity} + (-0.46) \times x_{color} = \text{Value}$$

$B_{intercept}$ B_{weight} $B_{clarity}$ B_{color}

Figure 15.2: A linear model estimating the values of diamonds based on their weight, clarity, and color.

```
## weight         2.189      0.200   10.95   <2e-16 ***
## clarity       21.692      2.143   10.12   <2e-16 ***
## color         -0.455      0.365   -1.25     0.21
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.7 on 146 degrees of freedom
## Multiple R-squared:  0.637,  Adjusted R-squared:  0.63
## F-statistic: 85.5 on 3 and 146 DF,  p-value: <2e-16
```

Here, we can see from the summary table that the model estimated β_{Int} (the intercept), to be 148.34, β_{weight} to be 2.19, $\beta_{clarity}$ to be 21.69, and , β_{color} to be -0.45. You can see the full linear model in Figure 15.2:

You can access lots of different aspects of the regression object. To see what's inside, use `names()`

```
# Which components are in the regression object?
names(diamonds.lm)
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

For example, to get the estimated coefficients from the model, just access the `coefficients` attribute:

```
# The coefficients in the diamond model
diamonds.lm$coefficients
## (Intercept)      weight     clarity       color
##    148.3354      2.1894     21.6922     -0.4549
```

If you want to access the entire statistical summary table of the coefficients, you just need to access them from the `summary` object:

```
# Coefficient statistics in the diamond model
summary(diamonds.lm)$coefficients
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept) 148.3354     3.6253  40.917 7.009e-82
## weight        2.1894     0.2000  10.948 9.706e-21
## clarity      21.6922     2.1429  10.123 1.411e-18
## color        -0.4549     0.3646  -1.248 2.141e-01
```

15.2.2 Getting model fits with fitted.values

To see the fitted values from a regression object (the values of the dependent variable predicted by the model), access the `fitted.values` attribute from a regression object with `$fitted.values`.

Here, I'll add the fitted values from the diamond regression model as a new column in the diamonds dataframe:

```
# Add the fitted values as a new column in the dataframe
diamonds$value.lm <- diamonds.lm$fitted.values

# Show the result
head(diamonds)
##   weight clarity color value value.lm weight.c clarity.c value.g190
## 1   9.35    0.88     4 182.5    186.1  -0.5511   -0.1196      FALSE
## 2  11.10    1.05     5 191.2    193.1   1.1989    0.0504       TRUE
## 3   8.65    0.85     6 175.7    183.0  -1.2511   -0.1496      FALSE
## 4  10.43    1.15     5 195.2    193.8   0.5289    0.1504       TRUE
## 5  10.62    0.92     5 181.6    189.3   0.7189   -0.0796      FALSE
## 6  12.35    0.44     4 182.9    183.1   2.4489   -0.5596      FALSE
##   pred.g190
## 1   0.16252
## 2   0.82130
## 3   0.03008
## 4   0.84559
## 5   0.44455
## 6   0.08688
```

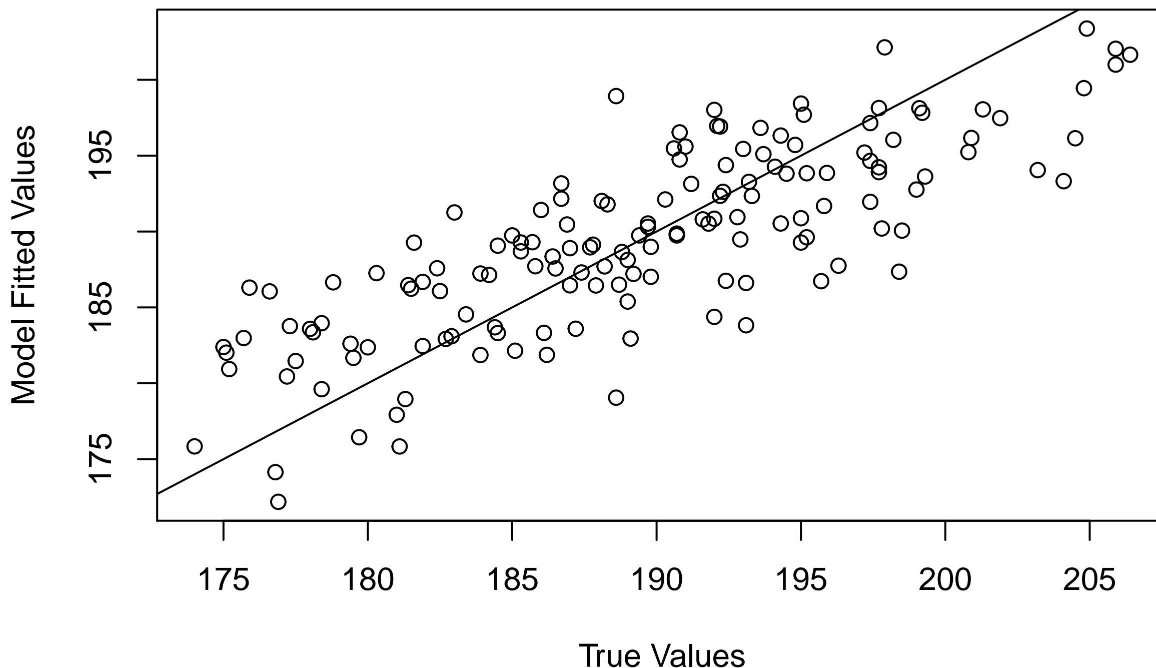
According to the model, the first diamond, with a weight of 9.35, a clarity of 0.88, and a color of 4 should have a value of 186.08. As we can see, this is not far off from the true value of 182.5.

You can use the fitted values from a regression object to plot the relationship between the true values and the model fits. If the model does a good job in fitting the data, the data should fall on a diagonal line:

```
# Plot the relationship between true diamond values
#   and linear model fitted values

plot(x = diamonds$value,                          # True values on x-axis
     y = diamonds.lm$fitted.values,               # fitted values on y-axis
     xlab = "True Values",
     ylab = "Model Fitted Values",
     main = "Regression fits of diamond values")

abline(b = 1, a = 0)                             # Values should fall around this line!
```

Table 15.2: 3 new diamonds

weight	clarity	color
20	1.5	5
10	0.2	2
15	5.0	3



15.2.3 Using `predict()` to predict new data from a model

Once you have created a regression model with `lm()`, you can use it to easily predict results from new datasets using the `predict()` function.

For example, let's say I discovered 3 new diamonds with the following characteristics:

I'll use the `predict()` function to predict the value of each of these diamonds using the regression model `diamond.lm` that I created before. The two main arguments to `predict()` are `object` – the regression object we've already defined), and `newdata` – the dataframe of new data. Warning! The dataframe that you use in the `newdata` argument to `predict()` must have column names equal to the names of the coefficients in the model. If the names are different, the `predict()` function won't know which column of data applies to which coefficient and will return an error.

```
# Create a dataframe of new diamond data
diamonds.new <- data.frame(weight = c(12, 6, 5),
                           clarity = c(1.3, 1, 1.5),
                           color = c(5, 2, 3))

# Predict the value of the new diamonds using
#  the diamonds.lm regression model
```

```
predict(object = diamonds.lm,      # The regression model
        newdata = diamonds.new)    # dataframe of new data
##     1     2     3
## 200.5 182.3 190.5
```

This result tells us the the new diamonds are expected to have values of 200.5, 182.3, and 190.5 respectively according to our regression model.

15.2.4 Including interactions in models: `y ~ x1 * x2`

To include interaction terms in a regression model, just put an asterix (*) between the independent variables. For example, to create a regression model on the diamonds data with an interaction term between `weight` and `clarity`, we'd use the formula `formula = value ~ weight * clarity`:

```
# Create a regression model with interactions between
#   IVS weight and clarity
diamonds.int.lm <- lm(formula = value ~ weight * clarity,
                      data = diamonds)

# Show summary statistics of model coefficients
summary(diamonds.int.lm)$coefficients
##                 Estimate Std. Error t value  Pr(>|t|)
## (Intercept)     157.4721     10.569 14.8987 4.170e-31
## weight            0.9787      1.070  0.9149 3.617e-01
## clarity           9.9245     10.485  0.9465 3.454e-01
## weight:clarity    1.2447      1.055  1.1797 2.401e-01
```

15.2.5 Center variables before computing interactions!

Hey what happened? Why are all the variables now non-significant? Does this mean that there is really no relationship between weight and clarity on value after all? No. Recall from your second-year pirate statistics class that when you include interaction terms in a model, you should always *center* the independent variables first. Centering a variable means simply subtracting the mean of the variable from all observations.

In the following code, I'll repeat the previous regression, but first I'll create new centered variables `weight.c` and `clarity.c`, and then run the regression on the interaction between these centered variables:

```
# Create centered versions of weight and clarity
diamonds$weight.c <- diamonds$weight - mean(diamonds$weight)
diamonds$clarity.c <- diamonds$clarity - mean(diamonds$clarity)

# Create a regression model with interactions of centered variables
diamonds.int.lm <- lm(formula = value ~ weight.c * clarity.c,
                      data = diamonds)

# Print summary
summary(diamonds.int.lm)$coefficients
##                     Estimate Std. Error t value   Pr(>|t|)
## (Intercept)          189.402     0.3831  494.39 2.908e-237
## weight.c               2.223     0.1988   11.18  2.322e-21
## clarity.c             22.248     2.1338   10.43  2.272e-19
## weight.c:clarity.c     1.245     1.0551    1.18  2.401e-01
```

Hey that looks much better! Now we see that the main effects are significant and the interaction is non-significant.

15.2.6 Getting an ANOVA from a regression model with `aov()`

Once you've created a regression object with `lm()` or `glm()`, you can summarize the results in an ANOVA table with `aov()`:

```
# Create ANOVA object from regression
diamonds.aov <- aov(diamonds.lm)

# Print summary results
summary(diamonds.aov)
##              Df Sum Sq Mean Sq F value Pr(>F)
## weight        1   3218    3218  147.40 <2e-16 ***
## clarity       1   2347    2347  107.53 <2e-16 ***
## color         1     34      34    1.56   0.21
## Residuals   146   3187      22
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

15.3 Comparing regression models with `anova()`

A good model not only needs to fit data well, it also needs to be parsimonious. That is, a good model should be only be as complex as necessary to describe a dataset. If you are choosing between a very simple model with 1 IV, and a very complex model with, say, 10 IVs, the very complex model needs to provide a much better fit to the data in order to justify its increased complexity. If it can't, then the more simpler model should be preferred.

To compare the fits of two models, you can use the `anova()` function with the regression objects as two separate arguments. The `anova()` function will take the model objects as arguments, and return an ANOVA testing whether the more complex model is significantly better at capturing the data than the simpler model. If the resulting p-value is sufficiently low (usually less than 0.05), we conclude that the more complex model is significantly better than the simpler model, and thus favor the more complex model. If the p-value is not sufficiently low (usually greater than 0.05), we should favor the simpler model.

Let's do an example with the diamonds dataset. I'll create three regression models that each predict a diamond's value. The models will differ in their complexity – that is, the number of independent variables they use. `diamonds.mod1` will be the simplest model with just one IV (weight), `diamonds.mod2` will include 2 IVs (weight and clarity) while `diamonds.mod3` will include three IVs (weight, clarity, and color).

```
# model 1: 1 IV (only weight)
 diamonds.mod1 <- lm(value ~ weight, data = diamonds)

 # Model 2: 2 IVs (weight AND clarity)
 diamonds.mod2 <- lm(value ~ weight + clarity, data = diamonds)

 # Model 3: 3 IVs (weight AND clarity AND color)
 diamonds.mod3 <- lm(value ~ weight + clarity + color, data = diamonds)
```

Now let's use the `anova()` function to compare these models and see which one provides the best parsimonious fit of the data. First, we'll compare the two simplest models: model 1 with model 2. Because these models differ in the use of the `clarity` IV (both models use `weight`), this ANVOA will test whether or not including the `clarity` IV leads to a significant improvement over using just the `weight` IV:

```
# Compare model 1 to model 2
anova(diamonds.mod1, diamonds.mod2)
## Analysis of Variance Table
##
## Model 1: value ~ weight
## Model 2: value ~ weight + clarity
##   Res.Df  RSS Df Sum of Sq   F Pr(>F)
## 1    148 5569
## 2    147 3221  1      2347 107 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As you can see, the result shows a Df of 1 (indicating that the more complex model has one additional parameter), and a very small p-value (< .001). This means that adding the `clarity` IV to the model *did* lead to a significantly improved fit over the model 1.

Next, let's use `anova()` to compare model 2 and model 3. This will tell us whether adding `color` (on top of weight and clarity) further improves the model:

```
# Compare model 2 to model 3
anova(diamonds.mod2, diamonds.mod3)
## Analysis of Variance Table
##
## Model 1: value ~ weight + clarity
## Model 2: value ~ weight + clarity + color
##   Res.Df  RSS Df Sum of Sq    F Pr(>F)
## 1    147 3221
## 2    146 3187  1        34 1.56   0.21
```

The result shows a non-significant result (p = 0.21). Thus, we should reject model 3 and stick with model 2 with only 2 IVs.

You don't need to compare models that only differ in one IV – you can also compare models that differ in multiple DVs. For example, here is a comparison of model 1 (with 1 IV) to model 3 (with 3 IVs):

```
# Compare model 1 to model 3
anova(diamonds.mod1, diamonds.mod3)
## Analysis of Variance Table
##
## Model 1: value ~ weight
## Model 2: value ~ weight + clarity + color
##   Res.Df  RSS Df Sum of Sq    F Pr(>F)
## 1    148 5569
## 2    146 3187  2      2381 54.5 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

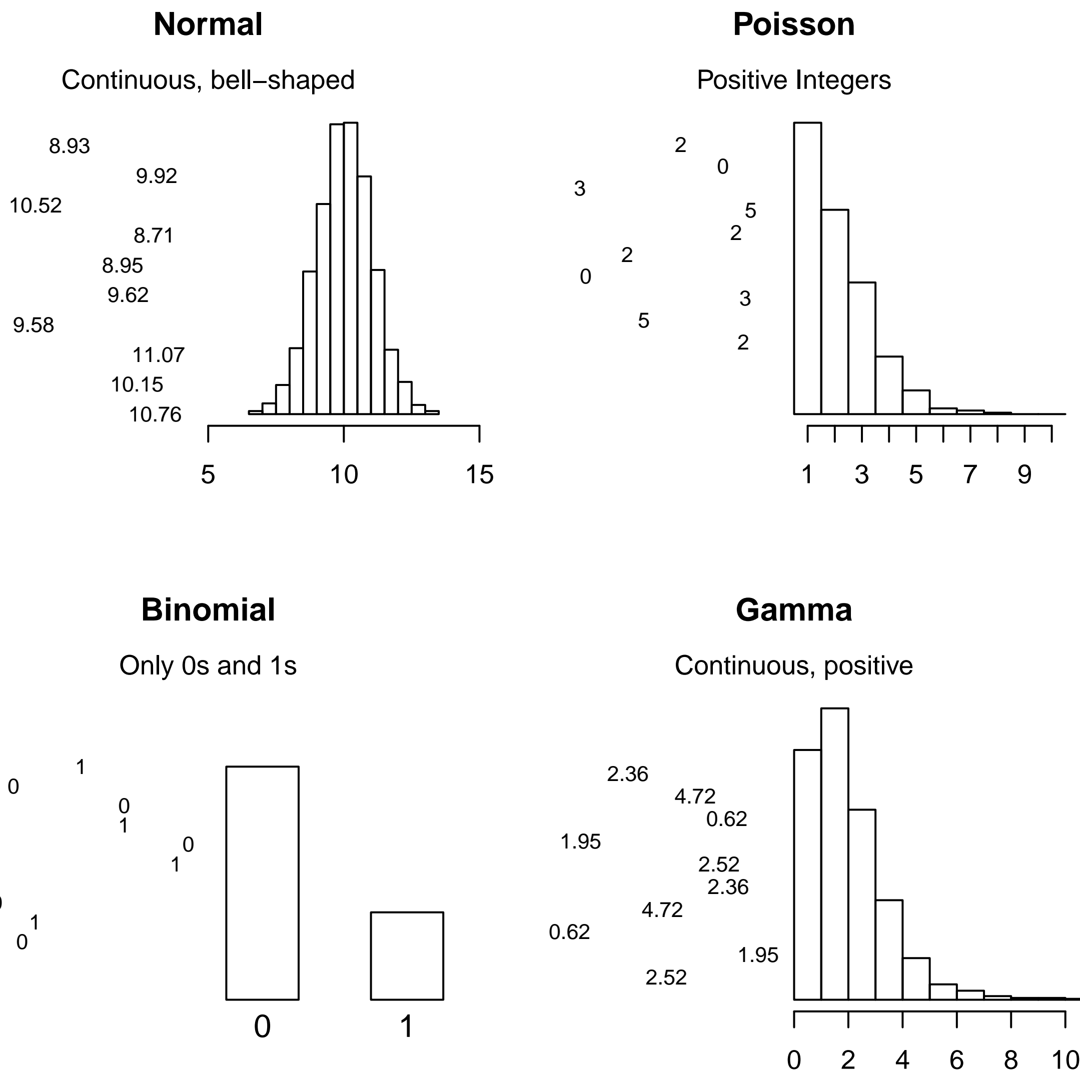
The result shows that model 3 did indeed provide a significantly better fit to the data compared to model 1. However, as we know from our previous analysis, model 3 is not significantly better than model 2.

15.4 Regression on non-Normal data with `glm()`

Argument	Description
`formula, data, subset`	The same arguments as in `lm()`
`family`	One of the following strings, indicating the link function for the general linear model

Family name	Description
`"binomial"`	Binary logistic regression, useful when the response is either 0 or 1.
`"gaussian"`	Standard linear regression. Using this family will give you the same result as `lm()`
`"Gamma"`	Gamma regression, useful for highly positively skewed data
`"inverse.gaussian"`	Inverse-Gaussian regression, useful when the dv is strictly positive and skewed to the right.
`"poisson"`	Poisson regression, useful for count data. For example, "How many parrots has a pirate owned over his/her lifetime¿'

We can use standard regression with `lm()`when your dependent variable is Normally distributed (more or less). When your dependent variable does not follow a nice bell-shaped Normal distribution, you need to use the *Generalized Linear Model* (GLM). the GLM is a more general class of linear models that change the distribution of your dependent variable. In other words, it allows you to use the linear model even when your dependent variable isn't a normal bell-shape. Here are 4 of the most common distributions you can can model with `glm()`:



15.5 Logistic regression with `glm(family = "binomial"`

The most common non-normal regression analysis is logistic regression, where your dependent variable is just 0s and 1. To do a logistic regression analysis with `glm()`, use the `family = binomial` argument.

Let's run a logistic regression on the diamonds dataset. First, I'll create a binary variable called `value.g190` indicating whether the value of a diamond is greater than 190 or not. Then, I'll conduct a logistic regression with our new binary variable as the dependent variable. We'll set `family = "binomial"` to tell `glm()` that the dependent variable is binary.

```
# Create a binary variable indicating whether or not
#   a diamond's value is greater than 190
diamonds$value.g190 <- diamonds$value > 190

# Conduct a logistic regression on the new binary variable
diamond.glm <- glm(formula = value.g190 ~ weight + clarity + color,
                   data = diamonds,
                   family = binomial)
```

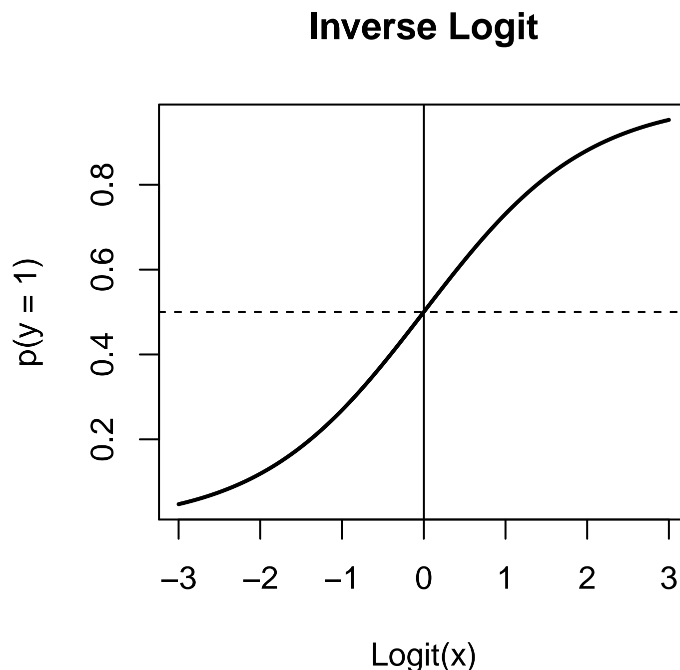


Figure 15.3: The inverse logit function used in binary logistic regression to convert logits to probabilities.

Here are the resulting coefficients:

```
# Print coefficients from logistic regression
summary(diamond.glm)$coefficients
##             Estimate Std. Error z value  Pr(>|z|)
## (Intercept) -18.8009     3.4634  -5.428 5.686e-08
## weight        1.1251     0.1968   5.716 1.088e-08
## clarity       9.2910     1.9629   4.733 2.209e-06
## color        -0.3836     0.2481  -1.547 1.220e-01
```

Just like with regular regression with `lm()`, we can get the fitted values from the model and put them back into our dataset to see how well the model fit the data:

```
# Add logistic fitted values back to dataframe as
#  new column pred.g190
diamonds$pred.g190 <- diamond.glm$fitted.values

# Look at the first few rows (of the named columns)
head(diamonds[c("weight", "clarity", "color", "value", "pred.g190")])
##   weight clarity color value pred.g190
## 1   9.35    0.88     4 182.5   0.16252
## 2  11.10    1.05     5 191.2   0.82130
## 3   8.65    0.85     6 175.7   0.03008
## 4  10.43    1.15     5 195.2   0.84559
## 5  10.62    0.92     5 181.6   0.44455
## 6  12.35    0.44     4 182.9   0.08688
```

Looking at the first few observations, it looks like the probabilities match the data pretty well. For example, the first diamond with a value of 182.5 had a fitted probability of just 0.16 of being valued greater than 190. In contrast, the second diamond, which had a value of 191.2 had a much higher fitted probability of 0.82.

Just like we did with regular regression, you can use the `predict()` function along with the results of a

glm() object to predict new data. Let's use the diamond.glm object to predict the probability that the new diamonds will have a value greater than 190:

```
# Predict the 'probability' that the 3 new diamonds
#  will have a value greater than 190

predict(object = diamond.glm,
        newdata = diamonds.new)
##       1       2       3
##  4.8605 -3.5265 -0.3898
```

What the heck, these don't look like probabilities! True, they're not. They are *logit-transformed* probabilities. To turn them back into probabilities, we need to invert them by applying the inverse logit function:

```
# Get logit predictions of new diamonds
logit.predictions <- predict(object = diamond.glm,
                             newdata = diamonds.new
                             )

# Apply inverse logit to transform to probabilities
#  (See Equation in the margin)
prob.predictions <- 1 / (1 + exp(-logit.predictions))

# Print final predictions!
prob.predictions
##       1       2       3
## 0.99231 0.02857 0.40376
```

So, the model predicts that the probability that the three new diamonds will be valued over 190 is 99.23%, 2.86%, and 40.38% respectively.

15.5.1 Adding a regression line to a plot

You can easily add a regression line to a scatterplot. To do this, just put the regression object you created with lm as the main argument to abline(). For example, the following code will create the scatterplot on the right (Figure~**??**) showing the relationship between a diamond's weight and its value including a red regression line:

```
# Scatterplot of diamond weight and value
plot(x = diamonds$weight,
     y = diamonds$value,
     xlab = "Weight",
     ylab = "Value",
     main = "Adding a regression line with abline()"
     )

# Calculate regression model
diamonds.lm <- lm(formula = value ~ weight,
                  data = diamonds)

# Add regression line
abline(diamonds.lm,
       col = "red", lwd = 2)
```

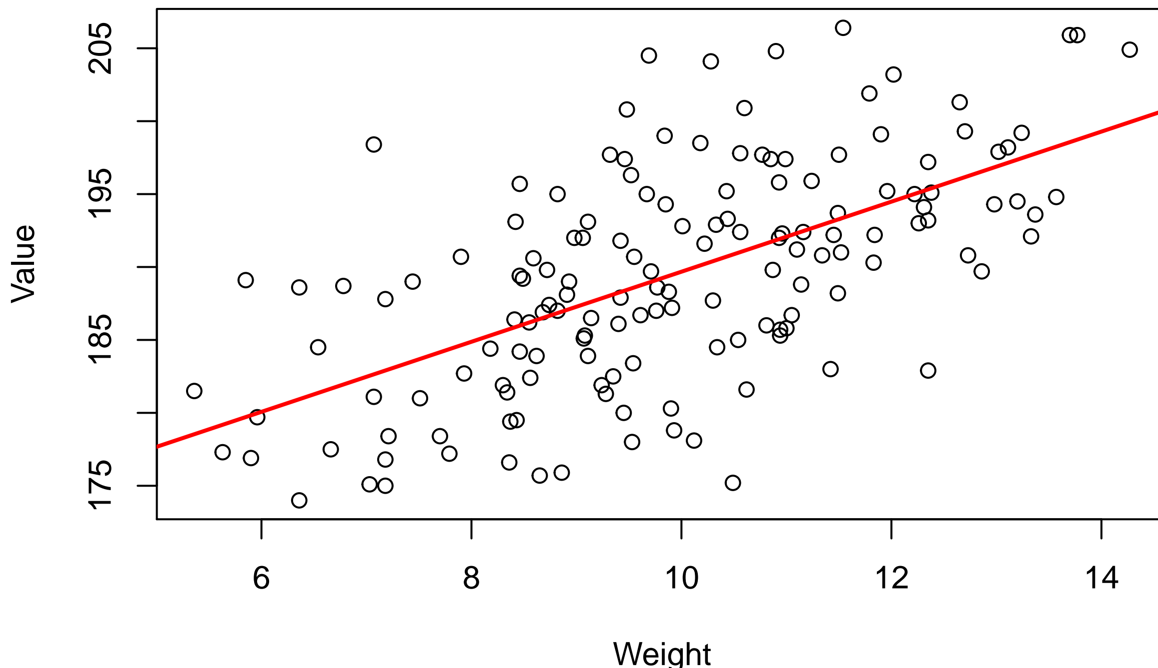


Figure 15.4: Adding a regression line to a scatterplot using abline()

15.5.2 Transforming skewed variables prior to standard regression

```
# The distribution of movie revenus is highly
#  skewed.
hist(movies$revenue.all,
     main = "Movie revenue\nBefore log-transformation")
```

If you have a highly skewed variable that you want to include in a regression analysis, you can do one of two things. Option 1 is to use the general linear model `glm()` with an appropriate family (like `family = "gamma"`). Option 2 is to do a standard regression analysis with `lm()`, but before doing so, transforming the variable into something less skewed. For highly skewed data, the most common transformation is a log-transformation.

For example, look at the distribution of movie revenues in the movies dataset in the margin Figure 15.5:

As you can see, these data don't look Normally distributed at all. There are a few movies (like Avatar) that just an obscene amount of money, and many movies that made much less. If we want to conduct a standard regression analysis on these data, we need to create a new log-transformed version of the variable. In the following code, I'll create a new variable called `revenue.all.log` defined as the logarithm of `revenue.all`

```
# Create a new log-transformed version of movie revenue
movies$revenue.all.log <- log(movies$revenue.all)
```

In Figure 15.6 you can see a histogram of the new log-transformed variable. It's still skewed, but not nearly as badly as before, so I would be feel much better using this variable in a standard regression analysis with `lm()`.

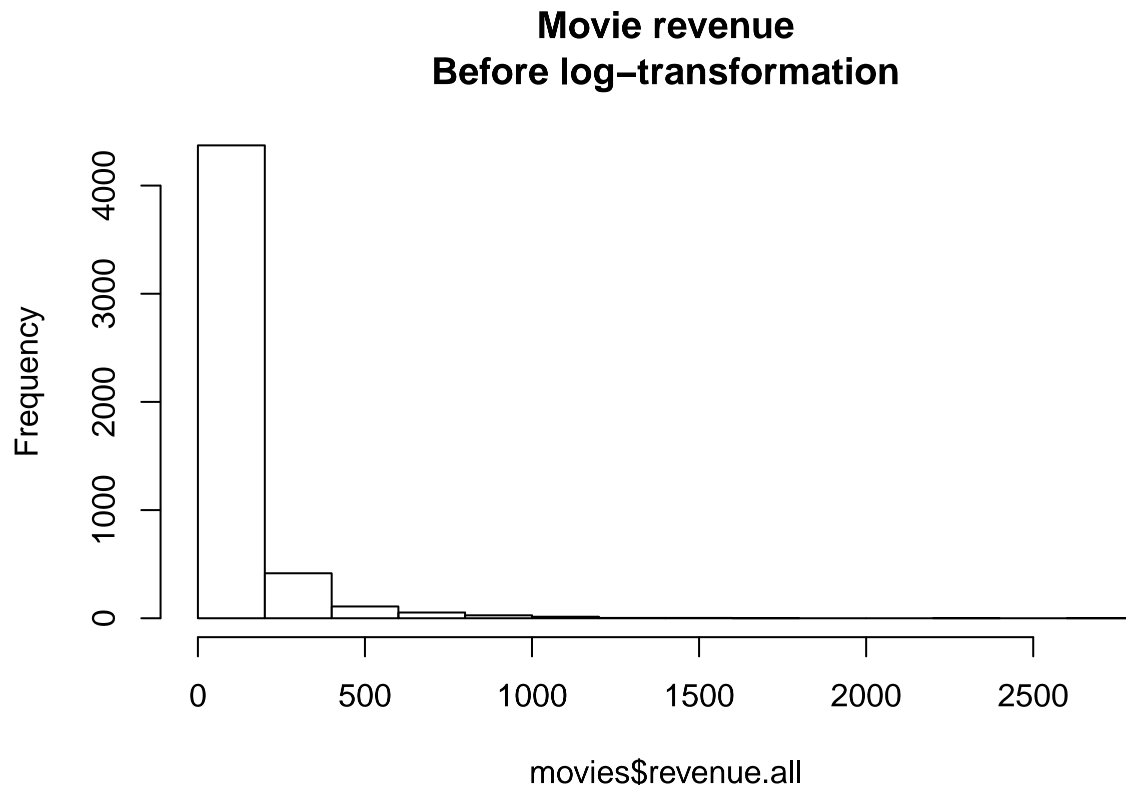


Figure 15.5: Distribution of movie revenues without a log-transformation

```
# Distribution of log-transformed
#  revenue is much less skewed

hist(movies$revenue.all.log,
     main = "Log-transformed Movie revenue")
```

15.6 Test your might! A ship auction

The following questions apply to the auction dataset in the yarrr package. This dataset contains information about 1,000 ships sold at a pirate auction. Here's how the first few rows of the dataframe should look:

```
head(auction)
##   cannons rooms age condition  color   style  jbb price price.gt.3500
## 1      18    20 140         5    red classic 3976  3502          TRUE
## 2      21    21  93         5    red  modern 3463  2955         FALSE
## 3      20    18  48         2   plum classic 3175  3281         FALSE
## 4      24    20  81         5 salmon classic 4463  4400          TRUE
## 5      20    21  93         2    red  modern 2858  2177         FALSE
## 6      21    19  60         6    red classic 4420  3792          TRUE
```

1. The column jbb is the "Jack's Blue Book" value of a ship. Create a regression object called `jbb.cannon.lm` predicting the JBB value of ships based on the number of cannons it has. Based on your result, how much value does each additional cannon bring to a ship?

Figure 15.6: Distribution of log-transformed movie revenues. It's still skewed, but not nearly as badly as before.

2. Repeat your previous regression, but do two separate regressions: one on modern ships and one on classic ships. Is there relationship between cannons and JBB the same for both types of ships?
3. Is there a significant interaction between a ship's style and its age on its JBB value? If so, how do you interpret the interaction?
4. Create a regression object called `jbb.all.lm` predicting the JBB value of ships based on cannons, rooms, age, condition, color, and style. Which aspects of a ship significantly affect its JBB value?
5. Create a regression object called `price.all.lm` predicting the actual selling value of ships based on cannons, rooms, age, condition, color, and style. Based on the results, does the JBB do a good job of capturing the effect of each variable on a ship's selling price?
6. Repeat your previous regression analysis, but instead of using the price as the dependent variable, use the binary variable *price.gt.3500* indicating whether or not the ship had a selling price greater than 3500. Call the new regression object `price.all.blr`. Make sure to use the appropriate regression function!!
7. Using `price.all.lm`, predict the selling price of the 3 new ships below

cannons	rooms	age	condition	color	style
12	34	43	7	black	classic
8	26	54	3	black	modern
32	65	100	5	red	modern

8. Using `price.all.blr`, predict the probability that the three new ships will have a selling price greater than 3500.

Chapter 16

Custom functions

16.1 Why would you want to write your own function?

Throughout this book, you have been using tons of functions either built into base-R – like `mean()`, `hist()`, `t.test()`, or written by other people and saved in packages – like `pirateplot()` and `apa()` in the `yarrr` package. However, because R is a complete programming language, you can easily write your *own* functions that perform specific tasks you want.

For example, let's say you think the standard histograms made with `hist()` are pretty boring. Instead, you'd like to you'd like use a fancier version with a more modern design that also displays statistical information. Now of course you know from an earlier chapter that you can customize plots in R any way that you'd like by adding customer parameter values like `col`, `bg` (etc.). However, it would be a pain to have to specify all of these custom parameters every time you want to create your custom histogram. To accomplish this, you can write your own custom function called `piratehist()` that automatically includes your custom specifications.

In the following code, I will define a new function called `piratehist()`. Just like the standard `hist()` function,`piratehist()` will take a vector of data (plus optional arguments indicated by `...`), create a light gray histogram, and adds text to the top of the figure indicating the mean and 95% CI of the data.

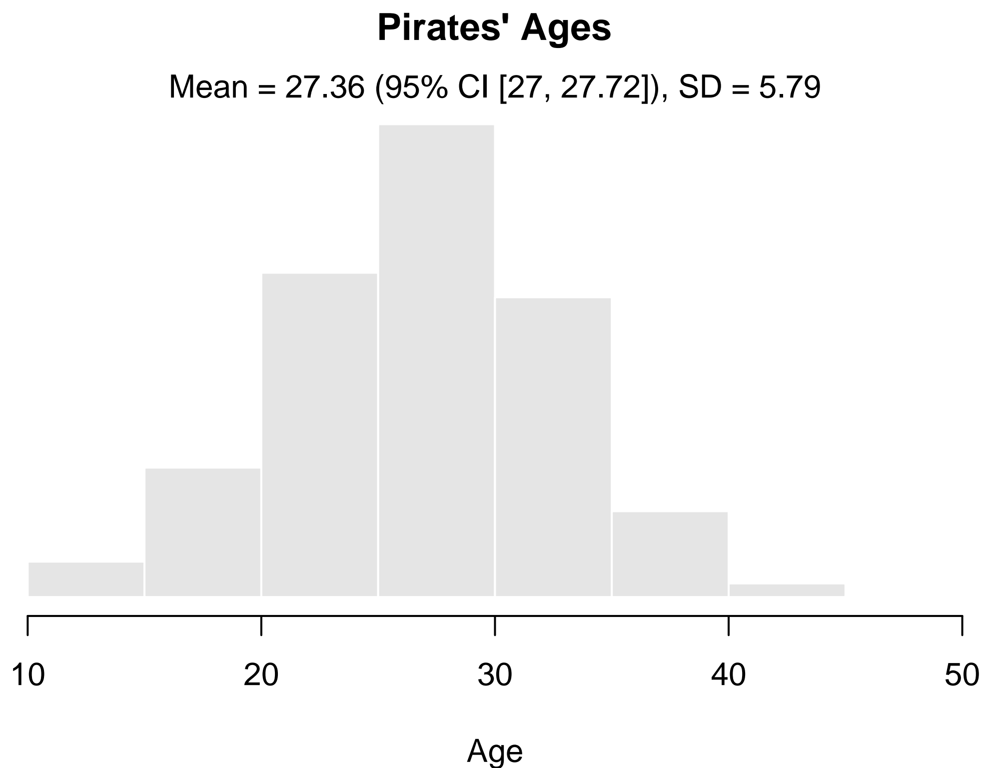
```
# Create a function called piratehist
piratehist <- function(x, ...) {

# Create a customized histogram
```

Figure 16.1: Functions. They're kind of a big deal.

```
hist(x,
     col = gray(.5, .2),
     border = "white",
     yaxt = "n",
     ylab = "",
     ...)

# Calculate the conf interval
ci <- t.test(x)$conf.int

# Define and add top-text
top.text <- paste(
  "Mean = ", round(mean(x), 2),
  " (95% CI [", round(ci[1], 2),
  ", ", round(ci[2], 2),
  "]), SD = ", round(sd(x), 2),
  sep = "")

mtext(top.text, side = 3)
}
```

Now that I've defined the `piratehist()` function, let's evaluate it on a vector of data!

```
# Create a pirate histogram!
piratehist(pirates$age,
        xlab = "Age",
        main = "Pirates' Ages")
```



As you can see, the resulting plot has all the customisations I specified in the function. So now, anytime I want to make a fancy pirate-y histogram, I can just use the `piratehist()` function rather than having to

always write all the raw code from scratch.

Of course, functions are limited to creating plots…oh no. You can write a function to do *anything* that you can program in R. Just think about a function as a container for R-code stored behind the scenes for you to use without having to see (or write) the code again. Now, if there's anything you like to do repeatedly in R (like making multiple customized plots, you can define the code just once in a new function rather than having to write it all again and again.Some of you reading this will quickly see how how writing your own functions can save you tons of time. For those of you who haven't…trust me, this is a big deal.

16.2 The structure of a custom function

A function is simply an object that (usually) takes some arguments, performs some action (executes some R code), and then (usually) returns some output. This might sound complicated, but you've been using functions pre-defined in R throughout this book. For example, the function `mean()` takes a numeric vector as an argument, and then returns the arithmetic mean of that vector as a single scalar value.

Your custom functions will have the following 4 attributes:

1. Name: What is the name of your function? You can give it any valid object name. However, be careful not to use names of existing functions or R might get confused.
2. Arguments: What are the inputs to the function? Does it need a vector of numeric data? Or some text? You can specify as many inputs as you want.
3. Actions: What do you want the function to do with the inputs? Create a plot? Calculate a statistic? Run a regression analysis? This is where you'll write all the real R code behind the function.
4. Output: What do you want the code to return when it's finished with the actions? Should it return a scalar statistic? A vector of data? A dataframe?

Here's how your function will look in R. When creating functions, you'll use two new functions (Yes, you use functions to create functions! Very Inception-y), called `function()` and `return()`. You'll put the function inputs as arguments to the `function()` function, and the output(s) as argument(s) to the `return()` function.

```
# The basic structure of a function
NAME <- function(ARGUMENTS) {

  ACTIONS

  return(OUTPUT)

}
```

16.2.1 Creating `my.mean()`

Let's create a custom functino called `my.mean()` that does the exact same thing as the `mean()` function in R. This function will take a vector `x` as an argument, creates a new vector called `output` that is the mean of all the elements of x (by summing all the values in x and dividing by the length of x), then return the `output` object to the user.

```
# Create the function my.mean()
my.mean <- function(x) {   # Single input called x

  output <- sum(x) / length(x) # Calculate output
```

```
return(output)  # Return output to the user after running the function

}
```

Try running the code above. When you do, nothing obvious happens. However, R has now stored the new function `my.mean()` in the current working directory for later use. To use the function, we can then just call our function like any other function in R. Let's call our new function on some data and make sure that it gives us the same result as `mean()`:

```
data <- c(3, 1, 6, 4, 2, 8, 4, 2)
my.mean(data)
## [1] 3.8
mean(data)
## [1] 3.8
```

As you can see, our new function `my.mean()` gave the same result as R's built in `mean()` function! Obviously, this was a bit of a waste of time as we simply recreated a built-in R function. But you get the idea...

16.2.2 Specifying multiple inputs

You can create functions with as many inputs as you'd like (even 0!). Let's do an example. We'll create a function called `oh.god.how.much.did.i.spend` that helps hungover pirates figure out how much gold they spent after a long night of pirate debauchery. The function will have three inputs: `grogg`: the number of mugs of grogg the pirate drank, `port`: the number of glasses of port the pirate drank, and `crabjuice`: the number of shots of fermented crab juice the pirate drank. Based on this input, the function will calculate how much gold the pirate spent. We'll also assume that a mug of grogg costs 1, a glass of port costs 3, and a shot of fermented crab juice costs 10.

```
oh.god.how.much.did.i.spend <- function(grogg,
                                        port,
                                        crabjuice) {

  output <- grogg * 1 + port * 3 + crabjuice * 10

  return(output)
}
```

Now let's test our new function with a few different values for the inputs grogg, port, and crab juice. How much gold did Tamara, who had had 10 mugs of grogg, 3 glasses of wine, and 0 shots of crab juice spend?

```
oh.god.how.much.did.i.spend(grogg = 10,
                            port = 3,
                            crabjuice = 0)
## [1] 19
```

Looks like Tamara spent 19 gold last night. Ok, now how about Cosima, who didn't drink any grogg or port, but went a bit nuts on the crab juice:

```
oh.god.how.much.did.i.spend(grogg = 0,
                            port = 0,
                            crabjuice = 7)
## [1] 70
```

Cosima's taste for crab juice set her back 70 gold pieces.

16.2.3 Including default values for arguments

When you create functions with many inputs, you'll probably want to start adding *default* values. Default values are input values which the function will use if the user does not specify their own. Most functions that you've used so far have default values. For example, the `hist()` function will use default values for inputs like `main`, `xlab`, (etc.) if you don't specify them/ Including defaults can save the user a lot of time because it keeps them from having to specify *every* possible input to a function.

To add a default value to a function input, just include `= DEFAULT}`after the input. For example, let's add a default value of 0 to each argument in the `oh.god.how.much.did.i.spend` function. By doing this, R will set any inputs that the user does not specify to 0 – in other words, it will assume that if you don't tell it how many drinks of a certain type you had, then you must have had 0.

```
# Including default values for function arguments
oh.god.how.much.did.i.spend <- function(grogg = 0,
                                         port = 0,
                                         crabjuice = 0) {

  output <- grogg * 1 + port * 3 + crabjuice * 10

  return(output)
}
```

Let's test the new version of our function with data from Hyejeong, who had 5 glasses of port but no grogg or crab juice. Because 0 is the default, we can just ignore these arguments:

```
oh.god.how.much.did.i.spend(port = 5)
## [1] 15
```

Looks like Hyejeong only spent 15 by sticking with port.

16.3 Using if, then statements in functions

A good function is like a person who knows what to wear for each occasion – it should put on different things depending on the occasion. In other words, rather than doing (i.e.; wearing) a tuxedo for every event, a good `dress()` function needs to first make sure that the input was (`event == "ball"`) rather than (`event == "jobinterview"`). To selectively evaluate code based on criteria, R uses *if-then* statements

To run an if-then statement in R, we use the `if() {}` function. The function has two main elements, a *logical test* in the parentheses, and *conditional code* in curly braces. The code in the curly braces is conditional because it is *only* evaluated if the logical test contained in the parentheses is `TRUE`. If the logical test is `FALSE`, R will completely ignore all of the conditional code.

Let's put some simple `if() {}` statements in a new function called `is.it.true()`. The function will take a single input `x`. If the input x is `TRUE`, the function will print one sentence. If the input x is `FALSE`, it will return a different sentence:

```
is.it.true <- function(x) {

if(x == TRUE) {print("x was true!")}
if(x == FALSE) {print("x was false!")}

}
```

Let's try evaluating the function on a few different inputs:

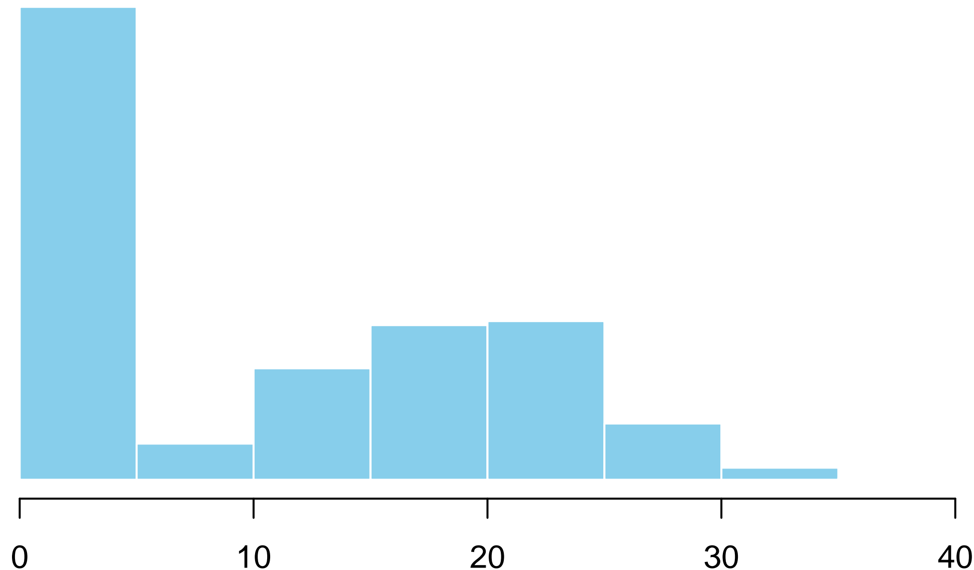
```
is.it.true(TRUE)
## [1] "x was true!"
is.it.true(FALSE)
## [1] "x was false!"
is.it.true(10 > 0)
## [1] "x was true!"
is.it.true(10 < 0)
## [1] "x was false!"
```

Using `if()` statements in your functions can allow you to do some really neat things. Let's create a function called `show.me()` that takes a vector of data, and either creates a plot, tells the user some statistics, or tells a joke! The function has two inputs: `x` – a vector of data, and `what` – a string value that tells the function what to do with x. We'll set the function up to accept three different values of `what` – either `"plot"`, which will plot the data, `"stats"`, which will return basic statistics about the vector, or `"tellmeajoke"`, which will return a funny joke!

```
show.me <- function(x, what) {

if(what == "plot") {

  hist(x, yaxt = "n", ylab = "", border = "white",
       col = "skyblue", xlab = "",
       main = "Ok! I hope you like the plot...")

}

if(what == "stats") {

  print(paste("Yarr! The mean of this data be ",
                round(mean(x), 2),
              " and the standard deviation be ",
              round(sd(x), 2),
              sep = ""))

}

if(what == "tellmeajoke") {

  print("I am a pirate, not your joke monkey.")

}
}
```

Let's try the `show.me()` function with different arguments:

```
show.me(x = pirates$beard.length,
        what = "plot")
```



Looks good! Now let's get the same function to tell us some statistics about the data by setting `what = "stats"`:

```
show.me(x = pirates$beard.length,
        what = "stats")
## [1] "Yarr! The mean of this data be 10.38 and the standard deviation be 10.31"
```

Phew that was exhausting, I need to hear a funny joke. Let's set `what = "tellmeajoke"`:

```
show.me(what = "tellmeajoke")
## [1] "I am a pirate, not your joke monkey."
```

That wasn't very funny.

16.4 A worked example: `plot.advanced()`

Let's create our own advanced own custom plotting function called `plot.advanced()` that acts like the normal plotting function, but has several additional arguments

1 `add.mean`: A logical value indicating whether or not to add vertical and horizontal lines at the mean value of x and y. 2 `add.regression`: A logical value indicating whether or not to add a linear regression line 3 `p.threshold`: A numeric scalar indicating the p.value threshold for determining significance 4 `add.modeltext`: A logical value indicating whether or not to include the regression equation as a sub-title to the plot

This plotting code is a bit long, but it's all stuff you've learned before.

```
plot.advanced <- function (x = rnorm(100),
                           y = rnorm(100),
                           add.mean = FALSE,
                           add.regression = FALSE,
                           p.threshold = .05,
                           add.modeltext = FALSE,
```

```
                           ...  # Optional further arguments passed on to plot
                           ) {

# Generate the plot with optional arguments
#   like main, xlab, ylab, etc.
  plot(x, y, ...)

# Add mean reference lines if add.mean is TRUE
if(add.mean == TRUE) {

  abline(h = mean(y), lty = 2)
  abline(v = mean(x), lty = 2)
}

# Add regression line if add.regression is TRUE
if(add.regression == TRUE) {

  model <- lm(y ~ x)  # Run regression

  p.value <- anova(model)$"Pr(>F)"[1] # Get p-value

  # Define line color from model p-value and threshold
  if(p.value < p.threshold) {line.col <- "red"}
  if(p.value >= p.threshold) {line.col <- "black"}

  abline(lm(y ~ x), col = line.col, lwd = 2) # Add regression line

}

  # Add regression equation text if add.modeltext is TRUE
if(add.modeltext == TRUE) {

  # Run regression
  model <- lm(y ~ x)

  # Determine coefficients from model object
  coefficients <- model$coefficients
  a <- round(coefficients[1], 2)
  b <- round(coefficients[2], 2)

  # Create text
  model.text <- paste("Regression Equation: ", a, " + ",
                      b, " * x", sep = "")

  # Add text to top of plot
  mtext(model.text, side = 3, line = .5, cex = .8)

}
}
```

Let's try it out!

```
plot.advanced(x = pirates$age,
              y = pirates$tchests,
```

```
              add.regression = TRUE,
              add.modeltext = TRUE,
              p.threshold = .05,
              main = "plot.advanced()",
              xlab = "Age", ylab = "Treasure Chests Found",
              pch = 16,
              col = gray(.2, .3))
```

plot.advanced()

Regression Equation: 0.21 + 0.82 * x

Treasure Chests Found

0 50 100 150

10 15 20 25 30 35 40 45

Age

16.4.1 Seeing function code

Because R is awesome, you can view the code underlying most functions by just evaluating the name of the function (without any parentheses or arguments). For example, the `yarrr` package contains a function called `transparent()` that converts standard colors into transparent colors. To see the code contained in the function, just evaluate its name:

```
# Show me the code in the transparent() function
transparent
## function(orig.col = "red", trans.val = 1, maxColorValue = 255) {
##
##   n.cols <- length(orig.col)
##   orig.col <- col2rgb(orig.col)
##
##   final.col <- rep(NA, n.cols)
##
##   for(i in 1:n.cols) {
##
##     final.col[i] <- rgb(orig.col[1, i], orig.col[2, i], orig.col[3, i],
##                         alpha = (1 - trans.val) * 255,
```

```
##                          maxColorValue = maxColorValue)
##
##   }
##
##   return(final.col)
## }
## <bytecode: 0x10e7e4ae0>
## <environment: namespace:yarrr>
```

Once you know the code underlying a function, you can easily copy it and edit it to your own liking. Or print it and put it above your bed. Totally up to you.

16.4.2 Using `stop()` to completely stop a function and print an error

By default, all the code in a function will be evaluated when it is executed. However, there may be cases where there's no point in evaluating some code and it's best to stop everything and leave the function altogether. For example, let's say you have a function called `do.stats()` that has a single argument called `mat` which is supposed to be a matrix. If the user accidentally enters a dataframe rather than a matrix, it might be best to stop the function altogether rather than to waste time executing code. To tell a function to stop running, use the `stop()` function.

If R ever executes a `stop()` function, it will automatically quit the function it's currently evaluating, and print an error message. You can define the exact error message you want by including a string as the main argument.

For example, the following function `do.stats` will print an error message if the argument `mat` is not a matrix.

```
do.stats <- function(mat) {

if(is.matrix(mat) == F) {stop("Argument was not a matrix!")}

# Only run if argument is a matrix!
print(paste("Thanks for giving me a matrix. The matrix has ", nrow(mat),
" rows and ", ncol(mat),
" columns. If you did not give me a matrix, the function would have stopped by now!",
sep = ""))

}
```

Let's test it. First I'll enter an argument that is definitely not a matrix:

```
do.stats(mat = "This is a string, not a matrix")
```

Now I'll enter a valid matrix argument:

```
do.stats(mat = matrix(1:10, nrow = 2, ncol = 5))
## [1] "Thanks for giving me a matrix. The matrix has 2 rows and 5 columns. If you did not give me a ma
```

16.4.3 Using vectors as arguments

You can use any kind of object as an argument to a function. For example, we could re-create the function `oh.god.how.much.did.i.spend` by having a single vector object as the argument, rather than three separate values. In this version, we'll extract the values of a, b and c using indexing:

```
oh.god.how.much.did.i.spend <- function(drinks.vec) {

  grogg <- drinks.vec[1]
  port <- drinks.vec[2]
  crabjuice <- drinks.vec[3]

  output <- grogg * 1 + port * 3 + crabjuice * 10

  return(output)

}
```

To use this function, the pirate will enter the number of drinks she had as a single vector with length three rather than as 3 separate scalars.

```
oh.god.how.much.did.i.spend(c(1, 5, 2))
## [1] 36
```

16.4.4 Storing and loading your functions to and from a function file with `source()`

As you do more programming in R, you may find yourself writing several function that you'll want to use again and again in many different R scripts. It would be a bit of a pain to have to re-type your functions every time you start a new R session, but thankfully you don't need to do that. Instead, you can store all your functions in one R file and then load that file into each R session.

I recommend that you put all of your custom R functions into a single R script with a name like `customfunctions.R`. Mine is called `Custom_Pirate_Functions.R`. Once you've done this, you can load all your functions into any R session by using the `source()` function. The source function takes a file directory as an argument (the location of your custom function file) and then executes the R script into your current session.

For example, on my computer my custom function file is stored at Users/Nathaniel/Dropbox/Custom_Pirate_Functions.R. When I start a new R session, I load all of my custom functions by running the following code:

```
# Evaluate all of the code in my custom function R script
source(file = "Users/Nathaniel/Dropbox/Custom_Pirate_Functions.R")
```

Once I've run this, I have access to all of my functions, I highly recommend that you do the same thing!

16.4.5 Testing functions

When you start writing more complex functions, with several inputs and lots of function code, you'll need to constantly test your function line-by-line to make sure it's working properly. However, because the input values are defined in the input definitions (which you won't execute when testing the function), you can't actually test the code line-by-line until you've defined the input objects in some other way. To do this, I recommend that you include temporary hard-coded values for the inputs at the beginning of the function code.

For example, consider the following function called `remove.outliers`. The goal of this function is to take a vector of data and remove any data points that are outliers. This function takes two inputs `x` and `outlier.def`, where `x` is a vector of numerical data, and `outlier.def` is used to define what an outlier is: if a data point is `outlier.def` standard deviations away from the mean, then it is defined as an outlier and is removed from the data vector.

In the following function definition, I've included two lines where I directly assign the function inputs to certain values (in this case, I set x to be a vector with 100 values of 1, and one outlier value of 999, and `outlier.def` to be 2). Now, if I want to test the function code line by line, I can uncomment these test values, execute the code that assigns those test values to the input objects, then run the function code line by line to make sure the rest of the code works.

```
remove.outliers <- function(x, outlier.def = 2) {

# Test values (only used to test the following code)
#  x <- c(rep(1, 100), 999)
#  outlier.def <- 2

  is.outlier <- x > (mean(x) + outlier.def * sd(x)) |
                x < (mean(x) - outlier.def * sd(x))

  x.nooutliers <- x[is.outlier == FALSE]

  return(x.nooutliers)

}
```

Trust me, when you start building large complex functions, hard-coding these test values will save you many headaches. Just don't forget to comment them out when you are done testing or the function will always use those values!

16.4.6 Using ... as a wildcard argument

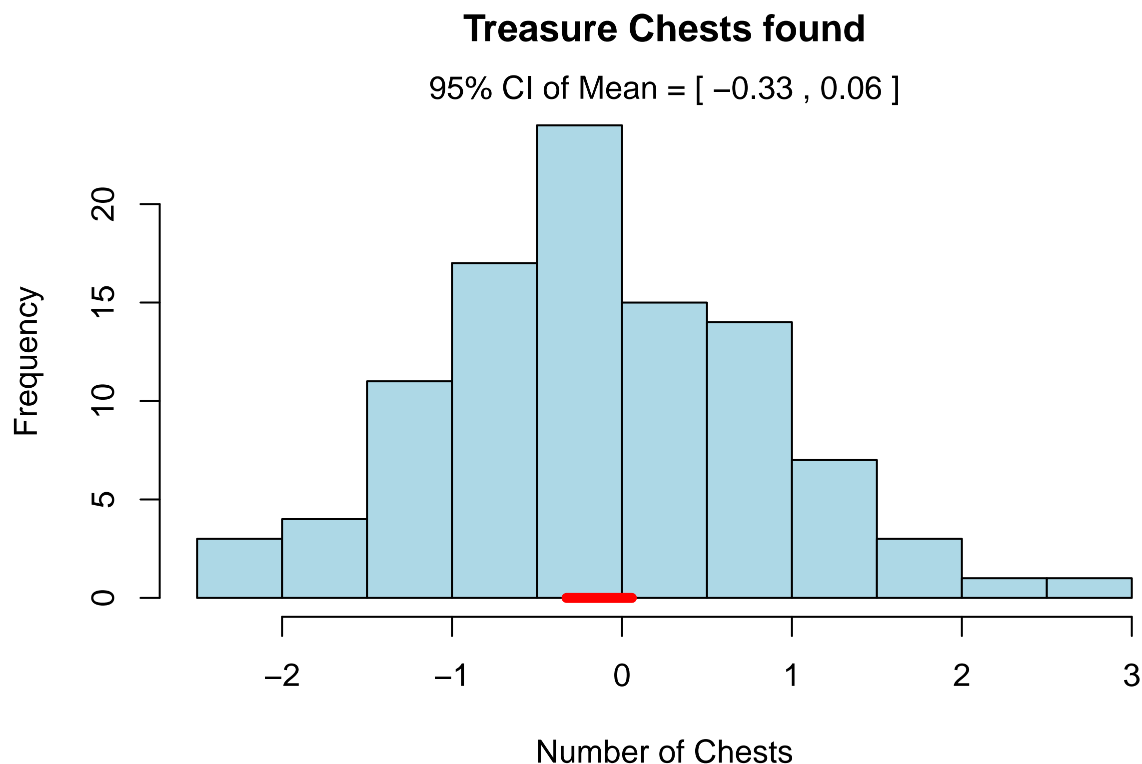
For some functions that you write, you may want the user to be able to specify inputs to functions within your overall function. For example, if I create a custom function that includes the histogram function `hist()` in R, I might also want the user to be able to specify optional inputs for the plot, like `main`, `xlab`, `ylab`, etc. However, it would be a real pain in the pirate ass to have to include all possible plotting parameters as inputs to our new function. Thankfully, we can take care of all of this by using the `...` notation as an input to the function. Note that the `...` notation will only pass arguments on to functions that are specifically written to allow for optional inputs. If you look at the help menu for `hist()`, you'll see that it does indeed allow for such option inputs passed on from other functions. The `...` input tells R that the user might add additional inputs that should be used later in the function.

Here's a quick example, let's create a function called `hist.advanced()` that plots a histogram with some optional additional arguments passed on with `...`

```
hist.advanced <- function(x, add.ci = TRUE, ...) {

hist(x, # Main Data
     ... # Here is where the additional arguments go
     )

if(add.ci == TRUE) {

  ci <- t.test(x)$conf.int # Get 95% CI
  segments(ci[1], 0, ci[2], 0, lwd = 5, col = "red")

  mtext(paste("95% CI of Mean = [", round(ci[1], 2), ",",
              round(ci[2], 2), "]"), side = 3, line = 0)
}
}
```

Now, let's test our function with the optional inputs `main`, `xlab`, and `col`. These arguments will be passed down to the `hist()` function within `hist.advanced()`. Here is the result:

```
hist.advanced(x = rnorm(100), add.ci = TRUE,
              main = "Treasure Chests found",
              xlab = "Number of Chests",
              col = "lightblue")
```



As you can see, R has passed our optional plotting arguments down to the main `hist()` function in the function code.

16.5 Test your R might!

1. Captain Jack is convinced that he can predict how much gold he will find on an island with the following equation: `(a * b) - c * 324 + log(a)`, where a is the area of the island in square meters, b is the number of trees on the island, and c is how drunk he is on a scale of 1 to 10. Create a function called `Jacks.Equation` that takes a, b, and c as arguments and returns Captain Jack's predictions. Here is an example of `Jacks.Equation` in action:

```
Jacks.Equation(a = 1000, b = 30, c = 7)
## [1] 27739
```

2. Write a function called `standardize.me` that takes a vector x as an argument, and returns a vector that standardizes the values of x (standardization means subtracting the mean and dividing by the standard deviation). Here is an example of `standardize.me` in action:

```
standardize.me(c(1, 2, 1, 100))
## [1] -0.51 -0.49 -0.51  1.50
```

3. Often times you will need to recode values of a dataset. For example, if you have a survey of age

data, you may want to convert any crazy values (like anything below 0 or above 100) to NA. Write a function called `recode.numeric()` with 3 arguments: `x`, `lb`, and `ub`. We'll assume that x is a numeric vector. The function should look at the values of x, convert any values below `lb` and above `ub` to NA, and then return the resulting vector. Here is the function in action:

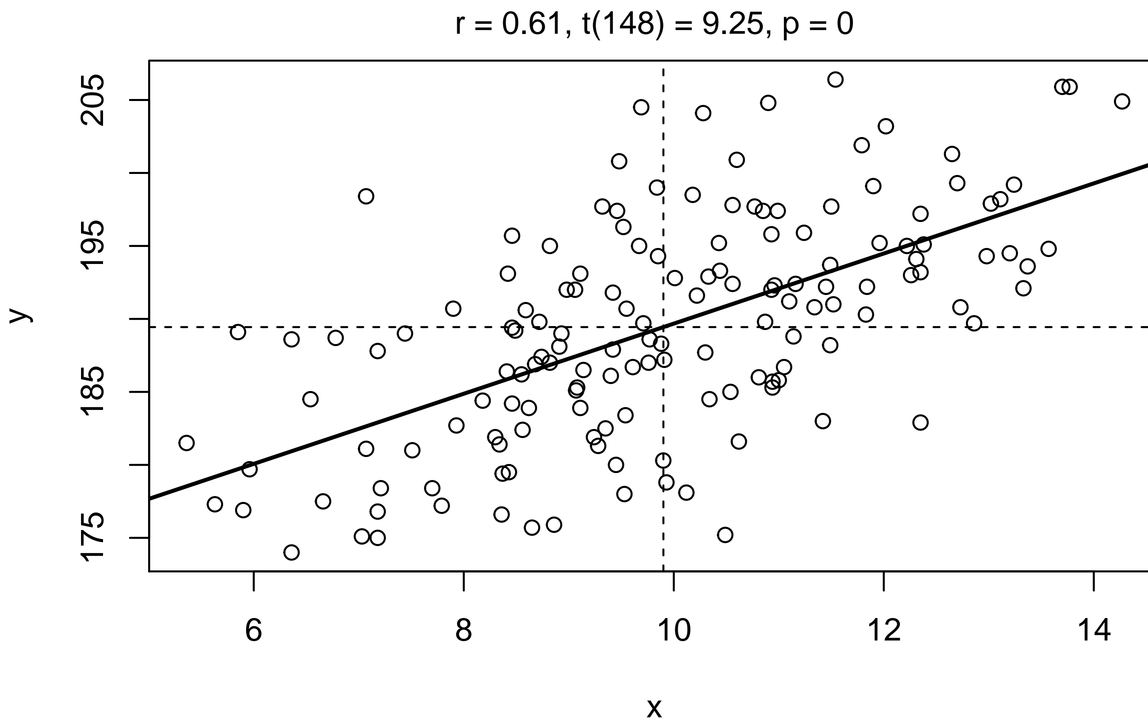
```
recode.numeric(x = c(5, 3, -5, 4, 3, 97),
               lb = 0,
               ub = 10)
## [1]  5  3 NA  4  3 NA
```

4. Create a function called `plot.advanced` that creates a scatterplot with the following arguments:

  - `add.regression`, a logical value indicating whether or not to add a regression line to the plot.
  - `add.means`, a logical value indicating whether or not to add a vertical line at the mean x value and a horizontal line at mean y value.
  - `add.test`, a logical value indicating whether or not to add text to the top margin of the plot indicating the result of a correlation test between x and y. (Hint: use `mtext()` and `paste()` to add the text)

Here is my version of `plot.advanced()` in action:

```
plot.advanced(x = diamonds$weight,
              y = diamonds$value,
              add.regression = TRUE,
              add.means = TRUE,
              add.test = TRUE)
```



Chapter 17

Loops

One of the golden rules of programming is D.R.Y. "Don't repeat yourself." Why? Not because you can't, but because it's almost certainly a waste of time. You see, while computers are still much, much worse than humans at some tasks (like recognizing faces), they are much, much better than humans at doing a few key things - like doing the same thing over…and over…and over. To tell R to do something over and over, we use a loop. Loops are absolutely critical in conducting many analyses because they allow you to write code once but evaluate it tens, hundreds, thousands, or millions of times without ever repeating yourself.

For example, imagine that you conduct a survey of 50 people containing 100 yes/no questions. Question 1 might be "Do you ever pick your nose?" and Question 2 might be "No seriously, do you ever pick your nose?!" When you finish the survey, you could store the data as a dataframe with 50 rows (one row for each person surveyed), and 100 columns representing all 100 questions. Now, because every question should have a yes or no answer, the only values in the dataframe should be "yes" or "no" Unfortunately, as is the case with all real world data collection, you will likely get some invalid responses – like "Maybe" or "What be yee phone number?!". For this reason, you'd like to go through all the data, and recode any invalid response as NA (aka, missing). To do this sequentially, you'd have to write the following 100 lines of code…

```
# SLOW way to convert any values that aren't equal to "Y", or "N" to NA
survey.df$q.1[(survey.data$q1 %in% c("Y", "N")) == FALSE] <- NA
survey.df$q.2[(survey.data$q2 %in% c("Y", "N")) == FALSE] <- NA
# . ... Wait...I have to type this 98 more times?!
# .
# . ... My god this is boring...
# .
survey.df$q.100[(survey.data$q100 %in% c("Y", "N")) == FALSE] <- NA
```

Pretty brutal right? Imagine if you have a huge dataset with 1,000 columns, now you're really doing a lot of typing. Thankfully, with a loop you can take care of this in no time. Check out this following code chunk which uses a loop to convert the data for *all* 100 columns in our survey dataframe.

```
# FAST way to convert values that aren't "Y", or "N" to NA

for(i in 1:100) { # Loop over all 100 columns

temp <- survey.df[, i]  # Get data for ith column and save in a new temporary object temp

temp[(temp %in% c("Y", "N")) == FALSE] <- NA # Convert invalid values in temp to NA

survey.df[, i] <- temp # Assign temp back to survey.df!
```

Figure 17.1: Loops in R can be fun. Just...you know...don't screw it up.

```
} # Close loop!
```

Done. All 100 columns. Take a look at the code and see if you can understand the general idea. But if not, no worries. By the end of this chapter, you'll know all the basics of how to construct loops like this one.

17.1 What are loops?

A loop is, very simply, code that tells a program like R to repeat a certain chunk of code several times with different values of a *loop object* that changes for every run of the loop. In R, the format of a for-loop is as follows:

```
# General structure of a loop
for(loop.object in loop.vector) {

  LOOP.CODE

  }
```

As you can see, there are three key aspects of loops: The *loop object*, the *loop vector*, and the *loop code*:

1. Loop object: The object that will change for each iteration of the loop. This is usually a letter like `i`, or an object with subscript like `column.i` or `participant.i`. You can use any object name that you want for the index. While most people use single character object names, sometimes it's more transparent to use names that tell you something about the data the object represents. For example, if you are doing a loop over participants in a study, you can call the index `participant.i`

2. Loop vector: A vector specifying all values that the loop object will take over the loop. You can specify the values any way you'd like (as long as it's a vector). If you're running a loop over numbers, you'll probably want to use `a:b` or `seq()`. However, if you want to run a loop over a few specific values, you can just use the `c()` function to type the values manually. For example, to run a loop over three different pirate ships, you could set the index values as `ship.i = c("Jolly Roger", "Black Pearl", "Queen Anne's Revenge")`.

3. Loop code: The code that will be executed for all values in the loop vector. You can write any R code you'd like in the loop code - from plotting to analyses. R will run this code for all possible values of the loop object specified in the loop vector.

17.1.1 Printing numbers from 1 to 100

Let's do a really simple loop that prints the integers from 1 to 10. For this code, our loop object is `i`, our loop vector is `1:10`, and our loop code is `print(i)`. You can verbally describe this loop as: *For every integer i between 1 and 10, print the integer i*:

```
# Print the integers from 1 to 10
for(i in 1:10) {

  print(i)

}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

As you can see, the loop applied the loop code (which in this case was `print(i)`) to every value of the loop object `i` specified in the `loop vector`, `1:10`.

17.1.2 Adding the integers from 1 to 100

Let's use a loop to add all the integers from 1 to 100. To do this, we'll need to create an object called `current.sum` that stores the latest sum of the numbers as we go through the loop. We'll set the loop object to `i`, the loop vector to `1:100`, and the loop code to `current.sum <- current.sum + i`. Because we want the starting sum to be 0, we'll set the initial value of `current.sum` to 0. Here is the code:

```
# Loop to add integers from 1 to 100

current.sum <- 0 # The starting value of current.sum

for(i in 1:100) {

 current.sum <- current.sum + i # Add i to current.sum

}

current.sum # Print the result!
## [1] 5050
```

Looks like we get an answer of 5050. To see if our loop gave us the correct answer, we can do the same calculation without a loop by using `a:b` and the `sum()` function:

Figure 17.2: Gauss. The guy was a total pirate. And totally would give us shit for using a loop to calculate the sum of 1 to 100...

```
# Add the integers from 1 to 100 without a loop
sum(1:100)
## [1] 5050
```

As you can see, the `sum(1:100)` code gives us the same answer as the loop (and is much simpler).

There's actually a funny story about how to quickly add integers (without a loop). According to the story, a lazy teacher who wanted to take a nap decided that the best way to occupy his students was to ask them to privately count all the integers from 1 to 100 at their desks. To his surprise, a young student approached him after a few moments with the correct answer: 5050. The teacher suspected a cheat, but the student didn't count the numbers. Instead he realized that he could use the formula `n(n+1) / 2`. Don't believe the story? Check it out:

```
# Calculate the sum of integers from 1 to 100 using Gauss' method
n <- 100
n * (n + 1) / 2
## [1] 5050
```

This boy grew up to be Gauss, a super legit mathematician.

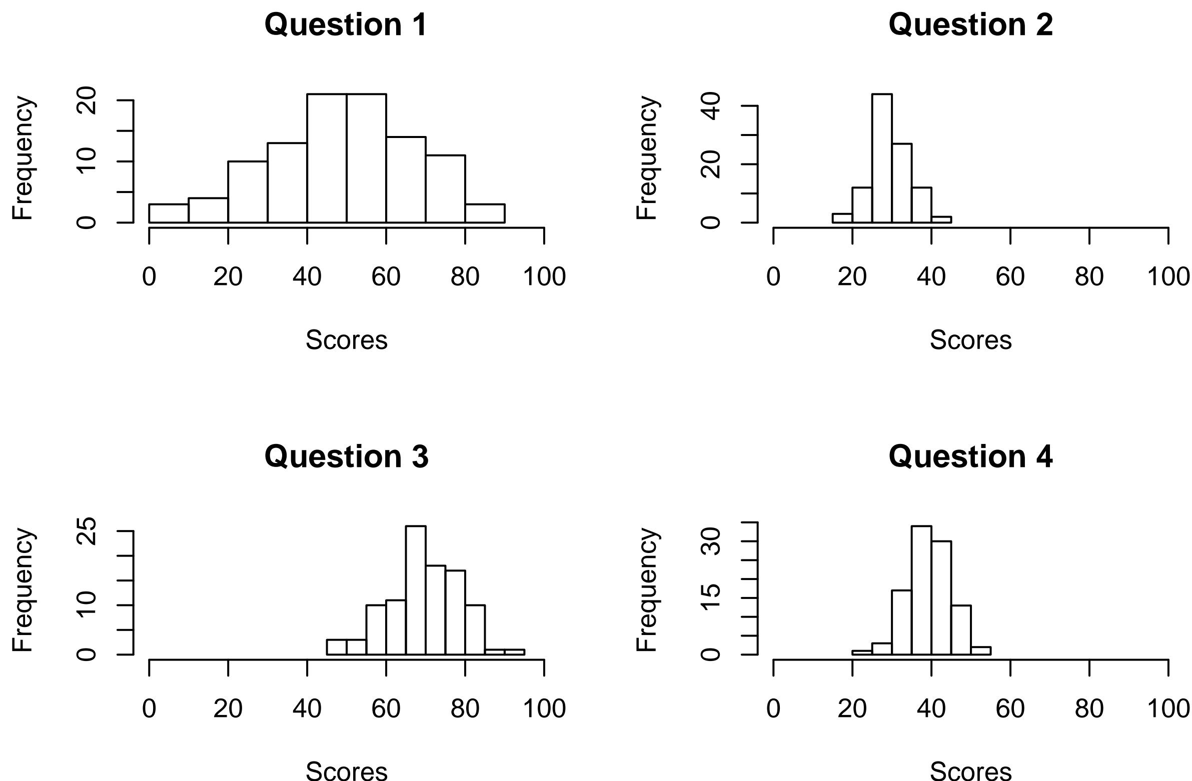
17.2 Creating multiple plots with a loop

One of the best uses of a loop is to create multiple graphs quickly and easily. Let's use a loop to create 4 plots representing data from an exam containing 4 questions. The data are represented in a matrix with 100 rows (representing 100 different people), and 4 columns representing scores on the different questions. The data are stored in the `yarrr` package in an object called `examscores`. Here are how the first few rows of the data look

```
# First few rows of the examscores data
head(examscores)
##    a  b  c  d
## 1 43 31 68 34
## 2 61 27 56 39
## 3 37 41 74 46
## 4 54 36 62 41
## 5 56 34 82 40
## 6 73 29 79 35
```

Now, we'll loop over the columns and create a histogram of the data in each column. First, I'll set up a 2 x 2 plotting space with `par(mfrow())` (If you haven't seen `par(mfrow())` before, just know that it allows you to put multiple plots side-by-side). Next, I'll define the `loop object` as `i`, and the `loop vector` as the integers from 1 to 4 with `1:4`. In the `loop code`, I stored the data in column `i` as a new vector `x`. Finally, I created a histogram of the object `x`!

```
par(mfrow = c(2, 2))  # Set up a 2 x 2 plotting space

# Create the loop.vector (all the columns)
loop.vector <- 1:4

for (i in loop.vector) { # Loop over loop.vector

  # store data in column.i as x
  x <- examscores[,i]

  # Plot histogram of x
  hist(x,
       main = paste("Question", i),
       xlab = "Scores",
       xlim = c(0, 100))
}
```



17.3 Updating a container object with a loop

For many loops, you may want to update values of a 'container' object with each iteration of a loop. We can easily do this using indexing and assignment within a loop.

Let's do an example with the `examscores` dataframe. We'll use a loop to calculate how many students failed each of the 4 exams – where failing is a score less than 50. To do this, we will start by creating an NA vector called `failure.percent`. This will be a container object that we'll update later with the loop.

```
# Create a container object of 4 NA values
failure.percent <- rep(NA, 4)
```

We will then use a loop that fills this object with the percentage of failures for each exam. The loop will go over each column in `examscores`, calculates the percentage of scores less than 50 for that column, and assigns the result to the ith value of `failure.percent`. For the loop, our loop object will be `i` and our loop vector will be `1:4`.

```
for(i in 1:4) { # Loop over columns 1 through 4

  # Get the scores for the ith column
  x <- examscores[,i]

  # Calculate the percent of failures
  failures.i <- mean(x < 50)

   # Assign result to the ith value of failure.percent
  failure.percent[i] <- failures.i

}
```

Figure 17.3: This is what I got when I googled "funny container".

Now let's look at the result.

```
failure.percent
## [1] 0.50 1.00 0.03 0.97
```

It looks like about 50% of the students failed exam 1, *everyone* (100%) failed exam 2, 3% failed exam 3, and 97% percent failed exam 4. To calculate `failure.percent` without a loop, we'd do the following:

```
# Calculate failure percent without a loop
failure.percent <- rep(NA, 4)
failure.percent[1] <- mean(examscores[,1] < 50)
failure.percent[2] <- mean(examscores[,2] < 50)
failure.percent[3] <- mean(examscores[,3] < 50)
failure.percent[4] <- mean(examscores[,4] < 50)
failure.percent
## [1] 0.50 1.00 0.03 0.97
```

As you can see, the results are identical.

17.4 Loops over multiple indices with a design matrix

So far we've covered simple loops with a single index value - but how can you do loops over multiple indices? You could do this by creating multiple nested loops. However, these are ugly and cumbersome. Instead, I recommend that you use `design matrices` to reduce loops with multiple index values into a single loop with just one index. Here's how you do it:

Let's say you want to calculate the mean, median, and standard deviation of some quantitative variable for all combinations of two factors. For a concrete example, let's say we wanted to calculate these summary statistics on the age of pirates for all combinations of colleges and sex.

To do this, we'll start by creating a design matrix. This matrix will have all combinations of our two factors. To create this design matrix matrix, we'll use the `expand.grid()` function. This function takes several vectors as arguments, and returns a dataframe with all combinations of values of those vectors. For our two factors college and sex, we'll enter all the factor values we want. Additionally, we'll add NA columns for the three summary statistics we want to calculate

```
design.matrix <- expand.grid("college" = c("JSSFP", "CCCC"), # college factor
                             "sex" = c("male", "female"), # sex factor
                             "median.age" = NA, # NA columns for our future calculations
                             "mean.age" = NA, #...
                             "sd.age" = NA, #...
                             stringsAsFactors = FALSE)
```

Here's how the design matrix looks:

```
design.matrix
##   college    sex median.age mean.age sd.age
## 1   JSSFP   male         NA       NA     NA
## 2    CCCC   male         NA       NA     NA
## 3   JSSFP female         NA       NA     NA
## 4    CCCC female         NA       NA     NA
```

As you can see, the design matrix contains all combinations of our factors in addition to three NA columns for our future statistics. Now that we have the matrix, we can use a single loop where the index is the row of the design.matrix, and the index values are all the rows in the design matrix. For each index value (that is, for each row), we'll get the value of each factor (college and sex) by indexing the current row of the

design matrix. We'll then subset the `pirates` dataframe with those factor values, calculate our summary statistics, then assign them

```
for(row.i in 1:nrow(design.matrix)) {

# Get factor values for current row
  college.i <- design.matrix$college[row.i]
  sex.i <- design.matrix$sex[row.i]

# Subset pirates with current factor values
  data.temp <- subset(pirates,
                      college == college.i & sex == sex.i)

# Calculate statistics
  median.i <- median(data.temp$age)
  mean.i <- mean(data.temp$age)
  sd.i <- sd(data.temp$age)

# Assign statistics to row.i of design.matrix
  design.matrix$median.age[row.i] <- median.i
  design.matrix$mean.age[row.i] <- mean.i
  design.matrix$sd.age[row.i] <- sd.i

}
```

Let's look at the result to see if it worked!

```
design.matrix
##   college    sex median.age mean.age sd.age
## 1   JSSFP   male         31       32    2.6
## 2    CCCC   male         24       23    4.3
## 3   JSSFP female         33       34    3.5
## 4    CCCC female         26       26    3.4
```

Sweet! Our loop filled in the NA values with the statistics we wanted.

17.5 The list object

Lists and loops go hand in hand. The more you program with R, the more you'll find yourself using loops.

Let's say you are conducting a loop where the outcome of each index is a vector. However, the length of each vector could change - one might have a length of 1 and one might have a length of 100. How can you store each of these results in one object? Unfortunately, a vector, matrix or dataframe might not be appropriate because their size is fixed. The solution to this problem is to use a `list()`. A list is a special object in R that can store virtually *anything*. You can have a list that contains several vectors, matrices, or dataframes of any size. If you want to get really Inception-y, you can even make lists of lists (of lists of lists....).

To create a list in R, use the `list()` function. Let's create a list that contains 3 vectors where each vector is a random sample from a normal distribution. We'll have the first element have 10 samples, the second will have 5, and the third will have 15.

```
# Create a list with vectors of different lengths
 number.list <- list(
      "a" = rnorm(n = 10),
      "b" = rnorm(n = 5),
```

```
      "c" = rnorm(n = 15))

 number.list
## $a
##  [1] -1.24  0.17 -1.19 -0.58 -1.23  0.92  1.62 -2.02 -0.25  0.92
##
## $b
## [1] -1.627 -1.103 -0.524  0.072 -2.351
##
## $c
##  [1] -0.6539  0.4062 -0.6126 -0.3552 -1.0043 -0.4276 -2.3236 -0.6905
##  [9] -0.4258 -1.0483  0.7000  0.7408 -0.0878  0.6970 -0.0016
```

To index an element in a list, use double brackets [[]] or $ if the list has names. For example, to get the first element of a list named `number.list`, we'd use `number.ls[[1]]`:

```
# Give me the first element in number.list
number.list[[1]]
##  [1] -1.24  0.17 -1.19 -0.58 -1.23  0.92  1.62 -2.02 -0.25  0.92

# Give me the element named b
number.list$b
## [1] -1.627 -1.103 -0.524  0.072 -2.351
```

Ok, now let's use the list object within a loop. We'll create a loop that generates 5 different samples from a Normal distribution with mean 0 and standard deviation 1 and saves the results in a list called `samples.ls`. The first element will have 1 sample, the second element will have 2 samples, etc.

First, we need to set up an empty list container object. To do this, use the `vector` function:

```
# Create an empty list with 5 elements
samples.ls <- vector("list", 5)
```

Now, let's run the loop. For each run of the loop, we'll generate `i` random samples and assign them to the ith element in `samples.ls`

```
 for(i in 1:5) {
   samples.ls[[i]] <- rnorm(n = i, mean = 0, sd = 1)
 }
```

Let's look at the result:

```
samples.ls
## [[1]]
## [1] 0.062
##
## [[2]]
## [1] 0.96 2.06
##
## [[3]]
## [1]  0.490  0.946 -0.023
##
## [[4]]
## [1]  1.45  0.28  0.67 -0.97
##
## [[5]]
## [1]  0.614 -0.098  0.778 -0.209  0.288
```

Looks like it worked. The first element has one sample, the second element has two samples and so on (you might get different specific values than I did because the samples were drawn randomly!).

17.6 Test your R might!

1. Using a loop, create 4 histograms of the weights of chickens in the `ChickWeight` dataset, with a separate histogram for time periods 0, 2, 4 and 6.
2. The following is a dataframe of survey data containing 5 questions I collected from 6 participants. The response to each question should be an integer between 1 and 5. Obviously, we have some invalid values in the dataframe. Let's fix them. Using a loop, create a new dataframe called `survey.clean` where all the invalid values (those that are not integers between 1 and 10) are set to NA.

```
survey <- data.frame("q1" = c(5, 3, 2, 7, 11, 5),
                     "q2" = c(4, 2, 2, 5, 5, 2),
                     "q3" = c(2, 1, 4, 2, 9, 10),
                     "q4" = c(2, 5, 2, 5, 4, 2),
                     "q5" = c(1, 4, -20, 2, 4, 2))
```

Here's how your `survey.clean` dataframe should look:

```
# The cleaned survey data
survey.clean
##   q1 q2 q3 q4 q5
## 1  5  4  2  2  1
## 2  3  2  1  5  4
## 3  2  2  4  2 NA
## 4  7  5  2  5  2
## 5 NA  5  9  4  4
## 6  5  2 10  2  2
```

3. Now, again using a loop, add a new column to the `survey.clean` dataframe called `invalid.answers` that indicates, for each participant, how many invalid answers they gave (Note: You may wish to use the `is.na()` function).
4. Standardizing a variable means subtracting the mean, and then dividing by the standard deviation. Using a loop, create a new dataframe called `survey.z` that contains standardized versions of the columns in the following `survey.B` dataframe.

```
survey.B <- data.frame("q1" = c(5, 3, 2, 7, 1, 9),
                     "q2" = c(4, 2, 2, 5, 1, 10),
                     "q3" = c(2, 1, 4, 2, 9, 10),
                     "q4" = c(10, 5, 2, 10, 4, 2),
                     "q5" = c(4, 4, 3, 2, 4, 2))
```

Here's how your `survey.B.z` dataframe should look:

```
survey.B.z
##      q1    q2    q3    q4    q5
## 1  0.16  0.00 -0.69  1.22  0.85
## 2 -0.49 -0.61 -0.94 -0.14  0.85
## 3 -0.81 -0.61 -0.17 -0.95 -0.17
## 4  0.81  0.30 -0.69  1.22 -1.19
## 5 -1.14 -0.91  1.12 -0.41  0.85
## 6  1.46  1.83  1.37 -0.95 -1.19
```

Chapter 18

Solutions

18.1 Chapter 4: The Basics

2. Which (if any) of the following objects names is/are invalid?

```
thisone <- 1
THISONE <- 2
1This <- 3
this.one <- 3
This.1 <- 4
ThIS.....ON...E <- 5
This!One! <- 6           # only this one!
lkjasdfkjsdf <- 7
```

3. 2015 was a good year for pirate booty - your ship collected 100,000 gold coins. Create an object called `gold.in.2015` and assign the correct value to it.

```
gold.in.2015 <- 100800
```

4. Oops, during the last inspection we discovered that one of your pirates Skippy McGee hid 800 gold coins in his underwear. Go ahead and add those gold coins to the object `gold.in.2015`. Next, create an object called `plank.list` with the name of the pirate thief.

```
gold.in.2015 <- gold.in.2015 + 800
plank.list <- "Skippy McGee"
```

5. Look at the code below. What will R return after the third line? Make a prediction, then test the code yourself.

```
a <- 10
a + 10
a       # It will return 10 because we never re-assigned a!
```

18.2 Chapter 5: Scalers and vectors

1. Create the vector [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] in three ways: once using `c()`, once using `a:b`, and once using `seq()`.

```
c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
##  [1]  1  2  3  4  5  6  7  8  9 10

1:10
##  [1]  1  2  3  4  5  6  7  8  9 10

seq(from = 1, to = 10, by = 1)
##  [1]  1  2  3  4  5  6  7  8  9 10
```

2. Create the vector [2.1, 4.1, 6.1, 8.1] in two ways, once using `c()` and once using `seq()`

```
c(2.1, 6.1, 6.1, 8.1)
## [1] 2.1 6.1 6.1 8.1

seq(from = 2.1, to = 8.1, by = 2)
## [1] 2.1 4.1 6.1 8.1
```

3. Create the vector [0, 5, 10, 15] in 3 ways: using `c()`, `seq()` with a `by` argument, and `seq()` with a `length.out` argument.

```
c(0, 5, 10, 15)
## [1]  0  5 10 15

seq(from = 0, to = 15, by = 5)
## [1]  0  5 10 15

seq(from = 0, to = 15, length.out = 4)
## [1]  0  5 10 15
```

4. Create the vector [101, 102, 103, 200, 205, 210, 1000, 1100, 1200] using a combination of the `c()` and `seq()` functions

```
c(seq(from = 101, to = 103, by = 3),
  seq(from = 200, to = 210, by = 5),
  seq(from = 1000, to = 1200, by = 100))
## [1]  101  200  205  210 1000 1100 1200
```

5. A new batch of 100 pirates are boarding your ship and need new swords. You have 10 scimitars, 40 broadswords, and 50 cutlasses that you need to distribute evenly to the 100 pirates as they board. Create a vector of length 100 where there is 1 scimitar, 4 broadswords, and 5 cutlasses in each group of 10. That is, in the first 10 elements there should be exactly 1 scimitar, 4 broadswords and 5 cutlasses. The next 10 elements should also have the same number of each sword (and so on).

```
swords <- rep(c("scimitar", rep("broadsword", 4), rep("cutlass", 5)), times = 100)
head(swords)
## [1] "scimitar"    "broadsword" "broadsword" "broadsword" "broadsword"
## [6] "cutlass"
```

6. Create a vector that repeats the integers from 1 to 5, 10 times. That is [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, …]. The length of the vector should be 50!

```
rep(1:5, times = 10)
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
## [36] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

7. Now, create the same vector as before, but this time repeat 1, 10 times, then 2, 10 times, etc., That is [1, 1, 1, …, 2, 2, 2, …, … 5, 5, 5]. The length of the vector should also be 50

Table 18.1: Renata's treasure haul when she was sober and when she was drunk

day	sober	drunk
Monday	2	0
Tuesday	0	0
Wednesday	3	1
Thursday	1	0
Friday	0	1
Saturday	3	2
Sunday	5	2

```
rep(1:5, each = 10)
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4
## [36] 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5
```

8. Create a vector containing 50 samples from a Normal distribution with a population mean of 20 and standard deviation of 2.

```
rnorm(n = 50, mean = 20, sd = 2)
##  [1] 18 22 23 17 22 22 17 22 20 21 22 23 24 21 22 23 23 20 21 18 21 23 18
## [24] 20 17 19 19 20 22 23 20 18 18 21 21 19 22 23 24 20 19 17 19 24 16 21
## [47] 22 18 18 23
```

9. Create a vector containing 25 samples from a Uniform distribution with a lower bound of -100 and an upper bound of -50.

```
runif(n = 25, min = -100, max = -50)
##  [1]  -65 -100  -53  -63  -92  -78  -75  -60  -76  -69  -77  -76  -73  -75
## [15]  -53  -60  -59  -96  -78  -61  -90  -77  -72  -91  -70
```

18.3 Chapter 6: Vector Functions

1. Create a vector that shows the square root of the integers from 1 to 10.

```
(1:10) ^ .5
##  [1] 1.0 1.4 1.7 2.0 2.2 2.4 2.6 2.8 3.0 3.2

#or

sqrt(1:10)
##  [1] 1.0 1.4 1.7 2.0 2.2 2.4 2.6 2.8 3.0 3.2
```

2. Renata thinks that she finds more treasure when she's had a mug of grogg than when she doesn't. To test this, she recorded how much treasure she found over 7 days without drinking any grogg (ie., sober), and then did the same over 7 days while drinking grogg (ie., drunk). Here are her results:

How much treasure did Renata find on average when she was sober? What about when she was drunk?

```
sober <- c(2, 0, 3, 1, 0, 3, 5)
drunk <- c(0, 0, 1, 0, 1, 2, 2)

mean(sober)
## [1] 2
```

```
mean(drunk)
## [1] 0.86
```

3. Using Renata's data again, create a new vector called `difference` that shows how much more treasure Renata found when she was drunk and when she was not. What was the mean, median, and standard deviation of the difference?

```
difference <- sober - drunk

mean(difference)
## [1] 1.1
median(difference)
## [1] 1
sd(difference)
## [1] 1.3
```

4. There's an old parable that goes something like this. A man does some work for a king and needs to be paid. Because the man loves rice (who doesn't?!), the man offers the king two different ways that he can be paid. *You can either pay me 100 kilograms of rice, or, you can pay me as follows: get a chessboard and put one grain of rice in the top left square. Then put 2 grains of rice on the next square, followed by 4 grains on the next, 8 grains on the next…and so on, where the amount of rice doubles on each square, until you get to the last square. When you are finished, give me all the grains of rice that would (in theory), fit on the chessboard.* The king, sensing that the man was an idiot for making such a stupid offer, immediately accepts the second option. He summons a chessboard, and begins counting out grains of rice one by one… Assuming that there are 64 squares on a chessboard, calculate how many grains of rice the main will receive. If one grain of rice weights 1/64000 kilograms, how many kilograms of rice did he get? *Hint: If you have trouble coming up with the answer, imagine how many grains are on the first, second, third and fourth squares, then try to create the vector that shows the number of grains on each square. Once you come up with that vector, you can easily calculate the final answer with the `sum()` function.*

```
# First, let's create a vector of the amount of rice on each square:
# It should be 1, 2, 4, 8, ...
rice <- 2 ^ (0:63)

# Here are the first few spaces
head(rice)
## [1]  1  2  4  8 16 32

# The result is just the sum!
rice.total <- sum(rice)
rice.total
## [1] 1.8e+19

# How much does that weigh? Each grain weights 1/6400 kilograms:
rice.kg <- sum(rice) * 1/6400
rice.kg
## [1] 2.9e+15

# That's 2,900,000,000,000,000 kilograms of rice. Let's keep going....
# A kg of rice is 1,300 calories

rice.cal <- rice.kg * 1300
rice.cal
## [1] 3.7e+18
```

Table 18.2: Some of my favorite movies

movie	year	boxoffice	genre	time	rating
Whatever Works	2009	35.0	Comedy	92	PG-13
It Follows	2015	15.0	Horror	97	R
Love and Mercy	2015	15.0	Drama	120	R
The Goonies	1985	62.0	Adventure	90	PG
Jiro Dreams of Sushi	2012	3.0	Documentary	81	G
There Will be Blood	2007	10.0	Drama	158	R
Moon	2009	321.0	Science Fiction	97	R
Spice World	1988	79.0	Comedy	-84	PG-13
Serenity	2005	39.0	Science Fiction	119	PG-13
Finding Vivian Maier	2014	1.5	Documentary	84	Unrated

```
# How many people can that feed for a year?
# A person needs about 2,250 calories a day, or 2,250 * 365 per year

rice.people.year <- rice.cal / (2250 * 365)
rice.people.year
## [1] 4.6e+12

# So, that amount of rice could feed 4,600,000,000,000 for a year
# Assuming that the averge lifespan is 70 years, how many lifespans could this feed?

rice.people.life <- rice.people.year / 70
rice.people.life
## [1] 6.5e+10

# Ok...so it could feed 65,000,000,000 (65 billion) people over their lives

# Conclusion: King done screwed up.
```

18.4 Chapter 7: Indexing vectors with []

0. Create new data vectors for each column.

```
movie <- c("Whatever Works", "It Follows", "Love and Mercy",
             "The Goonies", "Jiro Dreams of Sushi",
             "There Will be Blood", "Moon",
             "Spice World", "Serenity", "Finding Vivian Maier")

year <- c(2009, 2015, 2015, 1985, 2012, 2007, 2009, 1988, 2005, 2014)

boxoffice <- c(35, 15, 15, 62, 3, 10, 321, 79, 39, 1.5)

genre <- c("Comedy", "Horror", "Drama", "Adventure", "Documentary",
           "Drama", "Science Fiction", "Comedy", "Science Fiction",
           "Documentary")

time <- c(92, 97, 120, 90, 81, 158, 97, -84, 119, 84)
```

```
rating <- c("PG-13", "R", "R", "PG", "G", "R", "R",
            "PG-13", "PG-13", "Unrated")
```

1. What is the name of the 10th movie in the list?

```
movie[10]
## [1] "Finding Vivian Maier"
```

2. What are the genres of the first 4 movies?

```
genre[1:4]
## [1] "Comedy"    "Horror"    "Drama"     "Adventure"
```

3. Some joker put Spice World in the movie names – it should be "The Naked Gun" Please correct the name.

```
movie[movie == "Spice World"] <- "The Naked Gun"
```

4. What were the names of the movies made before 1990?

```
movie[year < 1990]
## [1] "The Goonies"   "The Naked Gun"
```

5. How many movies were Dramas? What percent of the 10 movies were Comedies?

```
sum(genre == "Drama")
## [1] 2

mean(genre == "Comedy")
## [1] 0.2
```

6. One of the values in the `time` vector is invalid. Convert any invalid values in this vector to NA. Then, calculate the mean movie time

```
time[time < 0] <- NA

mean(time, na.rm = TRUE)
## [1] 104
```

7. What were the names of the Comedy movies? What were their boxoffice totals? (Two separate questions)

```
movie[genre == "Comedy"]
## [1] "Whatever Works" "The Naked Gun"

boxoffice[genre == "Comedy"]
## [1] 35 79
```

8. What were the names of the movies that made less than $50 Million dollars AND were Comedies?

```
movie[boxoffice < 50 & genre == "Comedy"]
## [1] "Whatever Works"
```

9. What was the median boxoffice revenue of movies rated either G or PG?

```
median(boxoffice[rating %in% c("G", "PG")])
## [1] 32

# OR
```

```
median(boxoffice[rating == "G" | rating == "PG"])
## [1] 32
```

10. What percent of the movies were either rated R OR were comedies?

```
mean(rating == "R" | genre == "Comedy")
## [1] 0.6
```

18.5 Chapter 8: Matrices and Dataframes

The following table shows the results of a survey of 10 pirates. In addition to some basic demographic information, the survey asked each pirate "What is your favorite superhero?"" and "How many tattoos do you have?""

Name	Sex	Age	Superhero	Tattoos
Astrid	F	30	Batman	11
Lea	F	25	Superman	15
Sarina	F	25	Batman	12
Remon	M	29	Spiderman	5
Letizia	F	22	Batman	65
Babice	F	22	Antman	3
Jonas	M	35	Batman	9
Wendy	F	19	Superman	13
Niveditha	F	32	Maggott	900
Gioia	F	21	Superman	0

1. Combine the data into a single dataframe. Complete all the following exercises from the dataframe!

```
piratesurvey <- data.frame(
  name = c("Astrid", "Lea", "Sarina", "Remon", "Letizia", "Babice", "Jonas", "Wendy", "Niveditha", "Gio
  sex = c("F", "F", "F", "M", "F", "F", "M", "F", "F", "F"),
  age = c(30, 25, 25, 29, 22, 22, 35, 19, 32, 21),
  superhero = c("Batman", "Superman", "Batman", "Spiderman", "Batman",
                "Antman", "Batman", "Superman", "Maggott", "Superman"),
  tattoos = c(11, 15, 12, 5, 65, 3, 9, 13, 900, 0),
  stringsAsFactors = FALSE
)
```

2. What is the median age of the 10 pirates?

```
median(piratesurvey$age)
## [1] 25
```

3. What was the mean age of female and male pirates separately?

```
mean(piratesurvey$age[piratesurvey$sex == "F"])
## [1] 24
mean(piratesurvey$age[piratesurvey$sex == "M"])
## [1] 32

## OR
with(piratesurvey,
     mean(age[sex == "F"]))
## [1] 24
```

```
with(piratesurvey,
     mean(age[sex == "M"]))
## [1] 32

## OR

mean(subset(piratesurvey,
       subset = sex == "F")$age)
## [1] 24

mean(subset(piratesurvey,
       subset = sex == "M")$age)
## [1] 32
```

4. What was the most number of tattoos owned by a male pirate?

```
with(piratesurvey,
     max(tattoos[sex == "M"]))
## [1] 9

# OR

max(subset(piratesurvey,
           subset = sex == "M")$tattoos)
## [1] 9
```

5. What percent of pirates under the age of 32 were female?

```
with(piratesurvey,
     mean(sex[age < 32] == "F"))
## [1] 0.88
```

6. What percent of female pirates are under the age of 32?

```
with(piratesurvey,
     mean(sex[age < 32] == "F"))
## [1] 0.88
```

7. Add a new column to the dataframe called `tattoos.per.year` which shows how many tattoos each pirate has for each year in their life.

```
piratesurvey$tattoos.per.year <- with(piratesurvey, tattoos / age)
```

8. Which pirate had the most number of tattoos per year?

```
piratesurvey$name[piratesurvey$tattoos.per.year == max(piratesurvey$tattoos.per.year)]
## [1] "Niveditha"
```

9. What are the names of the female pirates whose favorite piratesurvey is Superman?

```
piratesurvey$name[with(piratesurvey, sex == "F" & superhero == "Superman")]
## [1] "Lea"   "Wendy" "Gioia"
```

10. What was the median number of tattoos of pirates over the age of 20 whose favorite piratesurvey is Spiderman?

```
with(piratesurvey, (tattoos[age > 20 & superhero == "Spiderman"]))
## [1] 5
```

18.6 Chapter 13: Hypothesis tests

1. Do male pirates have significantly longer beards than female pirates? Test this by conducting the appropriate test on the relevant data in the pirates dataset.

```
beard.sex.htest <- t.test(formula = beard.length ~ sex,
                          subset = sex %in% c("male", "female"),
                          data = pirates)

beard.sex.htest
##
##  Welch Two Sample t-test
##
## data:  beard.length by sex
## t = -70, df = 500, p-value <2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -20 -18
## sample estimates:
## mean in group female   mean in group male
##                  0.4                 19.4

apa(beard.sex.htest)
## [1] "mean difference = 19.02, t(499.82) = -70.89, p < 0.01 (2-tailed)"
```

Answer: Yes, men have significantly longer beards than women, mean difference = 19.02, t(499.82) = -70.89, p < 0.01 (2-tailed)

2. Are pirates whose favorite pixar movie is Up more or less likely to wear an eye patch than those whose favorite pixar movie is Inside Out? Test this by conducting the appropriate test on the relevant data in the pirates dataset.

```
df <- subset(pirates, fav.pixar %in% c("Up", "Inside Out"))
pixar.ep.table <- table(df$fav.pixar, df$eyepatch)

pixar.ep.htest <- chisq.test(pixar.ep.table)
pixar.ep.htest
##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  pixar.ep.table
## X-squared = 90, df = 1, p-value <2e-16

apa(pixar.ep.htest)
## [1] "X(1, N = 422) = 88.96, p < 0.01 (2-tailed)"
```

Answer: Yes, pirates whose favorite movie is Inside Out are much more likely to wear an eye patch than those whose favorite Pixar movie is Up, X(1, N = 422) = 88.96, p < 0.01 (2-tailed)

3. Do longer movies have significantly higher budgets than shorter movies? Answer this question by conducting the appropriate test in the movies dataset.

```
budget.time.htest <- cor.test(formula = ~ budget + time,
                              data = movies)

budget.time.htest
```

```
##
##  Pearson's product-moment correlation
##
## data:  budget and time
## t = 10, df = 2000, p-value <2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.24 0.32
## sample estimates:
##  cor
## 0.28

apa(budget.time.htest)
## [1] "r = 0.28, t(2313) = 14.09, p < 0.01 (2-tailed)"
```

Answer: Yes, longer movies tend to have higher budgets than shorter movies, r = 0.28, t(2313) = 14.09, p < 0.01 (2-tailed)

4. Do R rated movies earn significantly more money than PG-13 movies? Test this by conducting a the appropriate test on the relevant data in the movies dataset.

```
revenue.rating.htest <- t.test(formula = revenue.all ~ rating,
                               subset = rating %in% c("R", "PG-13"),
                               data = movies)

revenue.rating.htest
##
##  Welch Two Sample t-test
##
## data:  revenue.all by rating
## t = 10, df = 2000, p-value <2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  56 82
## sample estimates:
## mean in group PG-13     mean in group R
##                 148                  80

apa(revenue.rating.htest)
## [1] "mean difference = -68.86, t(1779.2) = 10.67, p < 0.01 (2-tailed)"
```

Answer: No, R Rated movies do not earn significantly more than PG-13 movies. In fact, PG-13 movies earn significantly more than R rated movies.

5. Are certain movie genres significantly more common than others in the movies dataset?

```
genre.table <- table(movies$genre)
genre.htest <- chisq.test(genre.table)

genre.htest
##
##  Chi-squared test for given probabilities
##
## data:  genre.table
## X-squared = 6000, df = 10, p-value <2e-16
```

```
apa(genre.htest)
## [1] "X(13, N = 4682) = 6408.91, p < 0.01 (2-tailed)"
```

Answer: Yes, some movie genres are more common than others, X(13, N = 4682) = 6408.91, p < 0.01 (2-tailed)

6. Do sequels and non-sequels differ in their ratings?

```
genre.sequel.table <- table(movies$genre, movies$sequel)

genre.sequel.htest <- chisq.test(genre.sequel.table)
## Warning in chisq.test(genre.sequel.table): Chi-squared approximation may be
## incorrect

apa(genre.sequel.htest)
## [1] "X(13, N = 4669) = 387.17, p < 0.01 (2-tailed)"
```

Answer: Yes, sequels are more likely in some genres than others.

Note: The error "Warning in chisq.test" we get in this code is due to the fact that some cells have no entries. This can make the test statistic unreliable. You can correct it by adding a value of 20 to every element in the table as follows:

```
genre.sequel.table <- table(movies$genre, movies$sequel)

# Add 20 to each cell to correct for empty cells
genre.sequel.table <- genre.sequel.table + 20

# Here is the result
genre.sequel.table
##
##                          0    1
##   Action               550  178
##   Adventure            384  141
##   Black Comedy          54   20
##   Comedy              1078  172
##   Concert/Performance   34   20
##   Documentary           83   20
##   Drama               1077   46
##   Horror               235  105
##   Multiple Genres       21   20
##   Musical               92   25
##   Reality               22   20
##   Romantic Comedy      265   23
##   Thriller/Suspense    425   41
##   Western               57   21

# Run a chi-square test on the table
genre.sequel.htest <- chisq.test(genre.sequel.table)

# Print the result
genre.sequel.htest
##
##  Pearson's Chi-squared test
##
## data:  genre.sequel.table
```

```
## X-squared = 400, df = 10, p-value <2e-16
```

18.7 Chapter 14: ANOVA

1. Is there a significant relationship between a pirate's favorite pixar movie and the number of tattoos (s)he has? Conduct an appropriate ANOVA with `fav.pixar` as the independent variable, and `tattoos` as the dependent variable. If there is a significant relationship, conduct a post-hoc test to determine which levels of the independent variable(s) differ.

```
pixar.aov <- aov(formula = tattoos ~ fav.pixar,
                 data = pirates)

summary(pixar.aov)
##              Df Sum Sq Mean Sq F value Pr(>F)
## fav.pixar    14    226    16.1    1.43   0.13
## Residuals   985  11105    11.3
```

Answer: No, there is no significant effect

2. Is there a significant relationship between a pirate's favorite pirate and how many tattoos (s)he has? Conduct an appropriate ANOVA with `favorite.pirate` as the independent variable, and `tattoos` as the dependent variable. If there is a significant relationship, conduct a post-hoc test to determine which levels of the independent variable(s) differ.

```
favpirate.aov <- aov(formula = tattoos ~ favorite.pirate,
                     data = pirates)

summary(favpirate.aov)
##                  Df Sum Sq Mean Sq F value Pr(>F)
## favorite.pirate   5     83    16.6    1.47    0.2
## Residuals       994  11248    11.3
```

Answer: No, there is no significant effect

3. Now, repeat your analysis from the previous two questions, but include both independent variables `fav.pixar` and `favorite.pirate` in the ANOVA. Do your conclusions differ when you include both variables?

```
pirpix.aov <- aov(formula = tattoos ~ favorite.pirate + fav.pixar,
                  data = pirates)

summary(pirpix.aov)
##                  Df Sum Sq Mean Sq F value Pr(>F)
## favorite.pirate   5     83    16.6    1.48   0.19
## fav.pixar        14    218    15.6    1.39   0.15
## Residuals       980  11029    11.2
```

4. Finally, test if there is an interaction between `fav.pixar` and `favorite.pirate` on number of tattoos.

```
pirpix.int.aov <- aov(formula = tattoos ~ favorite.pirate * fav.pixar,
                      data = pirates)

summary(pirpix.int.aov)
##                            Df Sum Sq Mean Sq F value Pr(>F)
## favorite.pirate             5     83    16.6    1.47   0.20
## fav.pixar                  14    218    15.6    1.38   0.16
```

```
## favorite.pirate:fav.pixar  65    685    10.5    0.93   0.63
## Residuals                 915  10344    11.3
```

Answer: Nope still nothing

18.8 Chapter 15: Regression

The following questions apply to the auction dataset in the yarrr package. This dataset contains information about 1,000 ships sold at a pirate auction.

1. The column jbb is the "Jack's Blue Book" value of a ship. Create a regression object called `jbb.cannon.lm` predicting the JBB value of ships based on the number of cannons it has. Based on your result, how much value does each additional cannon bring to a ship?

```
library(yarrr)

# jbb.cannon.lm model
# DV = jbb, IV = cannons
jbb.cannon.lm <- lm(formula = jbb ~ cannons,
                    data = auction)

# Print jbb.cannon.lm coefficients
summary(jbb.cannon.lm)$coefficients
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)     1396         61      23  1.3e-94
## cannons          101          3      34 1.9e-169
```

2. Repeat your previous regression, but do two separate regressions: one on modern ships and one on classic ships. Is there relationship between cannons and JBB the same for both types of ships?

```
# jbb.cannon.modern.lm  model
# DV = jbb, IV = cannons. Only include modern ships
jbb.cannon.modern.lm <- lm(formula = jbb ~ cannons,
                          data = subset(auction, style == "modern"))

# jbb.cannon.classic.lm model
# DV = jbb, IV = cannons. Only include classic ships
jbb.cannon.classic.lm <- lm(formula = jbb ~ cannons,
                          data = subset(auction, style == "classic"))

# Print jbb.cannon.modern.lm coefficients
summary(jbb.cannon.modern.lm)$coefficients
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)     1217       71.8      17  3.5e-51
## cannons          100        3.5      29 3.1e-107

# Print jbb.cannon.classic.lm coefficients
summary(jbb.cannon.classic.lm)$coefficients
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)     1537       75.9      20  5.9e-67
## cannons          104        3.7      28 2.0e-103
```

3. Is there a significant interaction between a ship's style and its age on its JBB value? If so, how do you interpret the interaction?

```
# int.lm model
# DV = jbb, IV = interaction between style and age
int.lm <- lm(formula = jbb ~ style * age,
             data = auction
             )

# Print int.lm coefficients
summary(int.lm)$coefficients
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)        3414.2      79.20   43.11 6.0e-230
## stylemodern         -15.7     111.74   -0.14  8.9e-01
## age                   1.9       0.76    2.57  1.0e-02
## stylemodern:age      -3.7       1.07   -3.43  6.2e-04
```

4. Create a regression object called `jbb.all.lm` predicting the JBB value of ships based on cannons, rooms, age, condition, color, and style. Which aspects of a ship significantly affect its JBB value?

```
# jbb.all.lm model
# DV = jbb, IV = everything (except price)]
jbb.all.lm <- lm(jbb ~ cannons + rooms + age + condition + color + style,
                 data = auction
                 )

# Print jbb.all.lm coefficients
summary(jbb.all.lm)$coefficients
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)     134.4       52.9    2.54  1.1e-02
## cannons         100.7        1.6   64.92  0.0e+00
## rooms            50.5        1.6   30.80 1.2e-146
## age               1.1        0.2    5.58  3.1e-08
## condition       107.6        3.9   27.51 3.4e-124
## colorbrown        4.9       16.6    0.30  7.7e-01
## colorplum       -29.8       31.3   -0.95  3.4e-01
## colorred         15.1       18.3    0.82  4.1e-01
## colorsalmon     -19.4       20.7   -0.94  3.5e-01
## stylemodern    -397.8       12.8  -30.98 6.7e-148
```

5. Create a regression object called `price.all.lm` predicting the actual selling value of ships based on cannons, rooms, age, condition, color, and style. Based on the results, does the JBB do a good job of capturing the effect of each variable on a ship's selling price?

```
# price.all.lm model
# DV = price, IV = everything (except jbb)]
price.all.lm <- lm(price ~ cannons + rooms + age + condition + color + style,
                 data = auction
                 )

# Print price.all.lm coefficients
summary(price.all.lm)$coefficients
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)     302.5      73.81    4.10  4.5e-05
## cannons         100.0       2.17   46.17 4.0e-249
## rooms            48.8       2.29   21.34  2.0e-83
## age               1.2       0.29    4.28  2.0e-05
## condition       104.1       5.46   19.05  3.4e-69
```

```
## colorbrown    -119.2      23.19   -5.14  3.3e-07
## colorplum       15.6      43.74    0.36  7.2e-01
## colorred      -603.6      25.59  -23.59  5.4e-98
## colorsalmon     70.4      28.97    2.43  1.5e-02
## stylemodern   -419.2      17.93  -23.38  1.3e-96
```

6. Repeat your previous regression analysis, but instead of using the price as the dependent variable, use the binary variable *price.gt.3500* indicating whether or not the ship had a selling price greater than 3500. Call the new regression object `price.all.blr`. Make sure to use the appropriate regression function!!.

```
# Create new binary variable indicating whether
#   a ship sold for more than 3500
auction$price.gt.3500 <- auction$price > 3500

# price.all.blr model
# DV = price.gt.3500, IV = everything (except jbb)
price.all.blr <- glm(price.gt.3500 ~ cannons + rooms + age + condition + color + style,
                  data = auction,
                  family = binomial   # Logistic regression
                  )

# price.all.blr coefficients
summary(price.all.blr)$coefficients
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -19.7401     1.4240  -13.86  1.1e-43
## cannons       0.6251     0.0442   14.14  2.2e-45
## rooms         0.2688     0.0296    9.07  1.2e-19
## age           0.0097     0.0033    2.93  3.4e-03
## condition     0.6825     0.0745    9.16  5.4e-20
## colorbrown   -0.8924     0.2549   -3.50  4.6e-04
## colorplum    -0.1291     0.5090   -0.25  8.0e-01
## colorred     -4.0764     0.4107   -9.93  3.2e-23
## colorsalmon   0.2479     0.3172    0.78  4.3e-01
## stylemodern  -2.4037     0.2432   -9.88  4.9e-23
```

7. Using `price.all.lm`, predict the selling price of the 3 new ships

cannons	rooms	age	condition	color	style
12	34	43	7	black	classic
8	26	54	3	black	modern
32	65	100	5	red	modern

```
# Create a dataframe with new ship data
new.ships <- data.frame(cannons = c(12, 8, 32),
                  rooms = c(34, 26, 65),
                  age = c(43, 54, 100),
                  condition = c(7, 3, 5),
                  color = c("black", "black", "red"),
                  style = c("classic", "modern", "modern"),
                  stringsAsFactors = FALSE)

# Predict new ship data based on price.all.lm model
predict(object = price.all.lm,
        newdata = new.ships
```

```
        )
##    1    2    3
## 3944 2331 6296
```

8. Using `price.all.blr`, predict the probability that the three new ships will have a selling price greater than 3500.

```
# Calculate logit of predictions
log.pred <- predict(object = price.all.blr,
                    newdata = new.ships
                    )

# Convert logits to probabilities
1 / (1 + exp(-log.pred))
##       1       2       3
## 0.89038 0.00051 1.00000
```

Bibliography