YaRrr!

The Pirate's Guide to R

DR. NATHANIEL D. PHILLIPS

YARRR! THE PIRATE'S GUIDE TO R

PUBLISHED BY

http://www.thepiratesguidetor.com

First printing,

Contents

This book is dedicated to Dr. Thomas Moore and Dr. Wei Lin who taught me everything I know about statistics, Dr. Dirk Wulff who taught me everything I know about R, and Dr. Hansjörg Neth who did the same with LaTeX.

Introduction

Who am I?

My name is Nathaniel. 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 www.nathanieldphillips.com. As you will learn in a minute, I didn't write this book, but I did translate it from pirate-speak to English.

Buy me a mug of beer!

This book is totally free. You are welcome to share it with your friends, family, hairdresser, plumber, and other loved ones. If you like it, and want to support me in improving the book, consider buying me a mug of beer with a donation. You can find a donation link at http://www.thepiratesguidetor.com.

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

Where did this book come from?

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.

This book is in progress..

This book is very much a work in progress and was last updated on . As I am still improving the pirate-English translations, there are likely many typos, errors and omissions in the document. I'm constantly experimenting with the material and the layout. If you have any recommendations for changes or spot any errors, please write me at YaRrr.Book@gmail.com or tweet me @YaRrrBook.

Email me with comments, recommendations or typos at: YaRrr.Book@gmail.com or tweet me at @YaRrrBook

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!

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.

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

2. R is the future. 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.

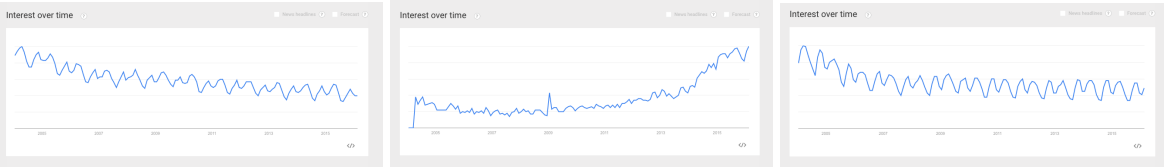


Figure 1: I wonder which trend is for R...

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 in Figure 2. If you can imagine an analytical task, you can almost certainly implement it in R.

4. Using RStudio, 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.

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 kitchen. 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.

Additional Tips

Because this is a beginner's book, I try to avoid bombarding you with too many details and tips when I first introduce a function. For this reason, at the end of every chapter, I present several tips and tricks that I think most R users should know once they get comfortable with the basics. I highly encourage you to read these additional tips as I expect you'll find at least some of them very useful if not invaluable.

```
library("circlize")
mat = matrix(sample(1:100, 18, replace = TRUE), 3, 6)
rownames(mat) = letters[1:3]
colnames(mat) = LETTERS[1:6]
chordDiagram(mat)
```

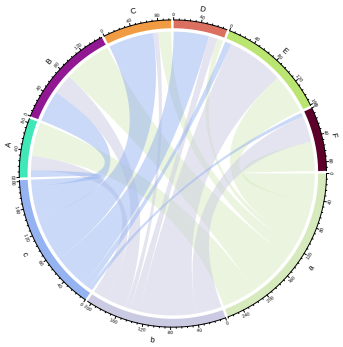


Figure 2: This is a `chordDiagram` plot that comes with the R package `circlize`. How cool is this?! Have fun trying to make this with SPSS.

1: Getting Started (and why R is like a relationship)

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!

So by now you've installed R and you're ready to get started. But first, let me give you a brief word of warning: Especially if this is your first experience programming, you are going to experience a *lot* of headaches when 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."

This is perfectly normal! Don't get discouraged and DON'T GO BACK TO SPSS! 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.

Figure 3: Yep, R will become both your best friend and your worst nightmare. The bad times will make the good times oh so much sweeter.

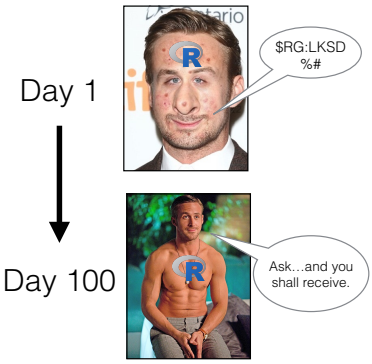


Figure 4: When you first meet R, it will look so fugly that you'll wonder if this is all some kind of sick joke. But trust me, once you learn how to talk to it, and clean it up a bit, all your friends will be crazy jealous.

Fun fact: SPSS stands for "Shitty Piece of Shitty Shit". True story.

Installing R and RStudio

First things first, let's download both Base R and Rstudio. Of course, they are totally free and open source.

Download Base R

- Windows: http://cran.r-project.org/bin/windows/base/
- 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 5 (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.

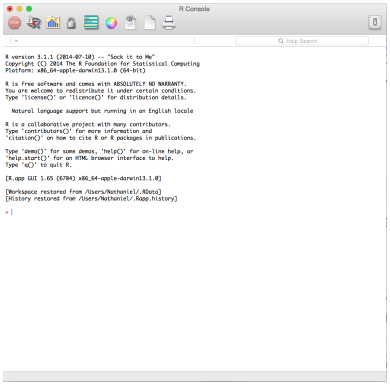


Figure 5: Here is how the standard R application looks. Not too exciting - just how we like it!

Download RStudio

- Windows and Mac: http://www.rstudio.com/products/rstudio/download/

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.

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

Code Chunks

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

```
a <- 1 + 2 + 3 + 4 + 5
a

## [1] 15
```

This is called a *code chunk*. You should always be able to directly 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. As you'll soon learn, lines that begin with # are either comments or output from prior code that R will ignore.

As you'll notice, I'll include code chunks before all plots in the book. In early chapters, the code might not make sense just yet. However, I elected to always include plotting code so you have the option of re-creating (and tweaking) any plot in the book.

The four RStudio windows

When you open RStudio, you'll see the following four windows (also called panes) shown in in Figure 6. However, your windows might be in a different order that those in Figure 6. 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.

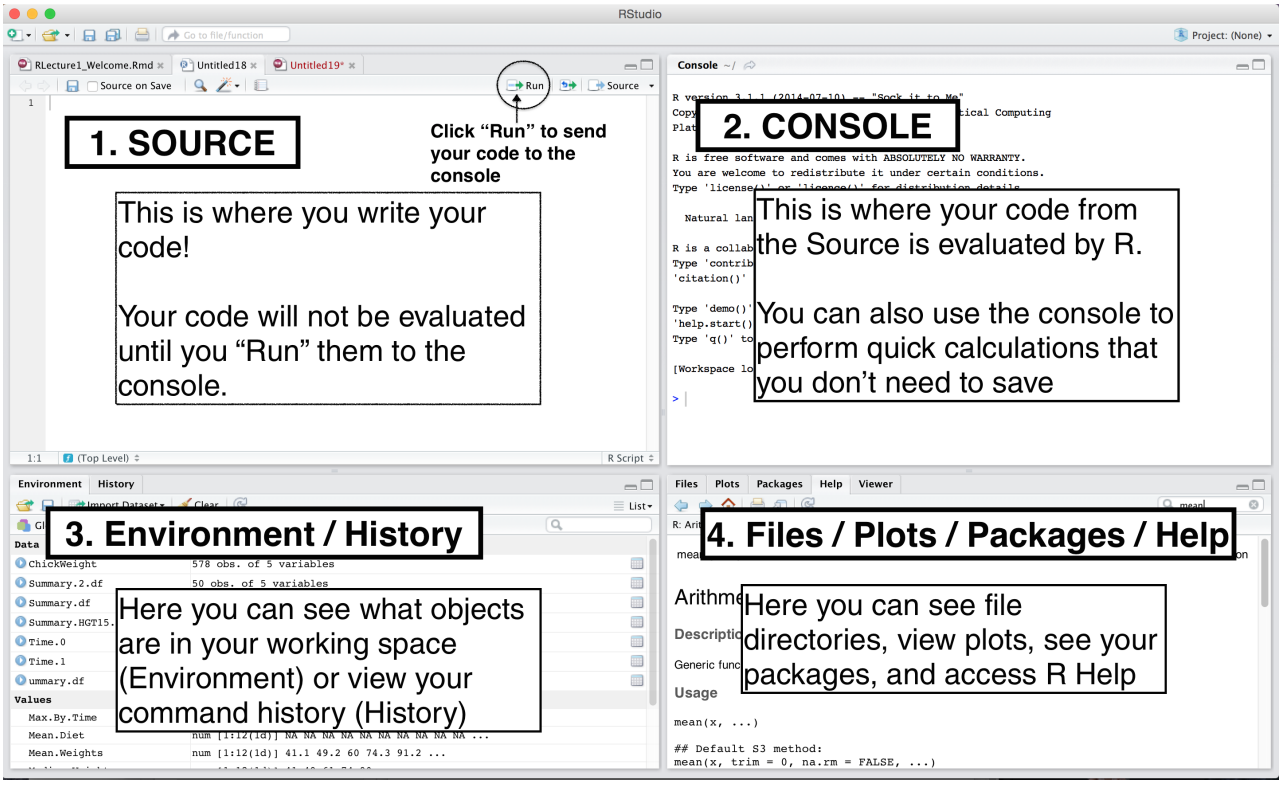


Figure 6: The four panes of RStudio.

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

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.

Console: The calculator

The console 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.

```
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.

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%

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.

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.

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.

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.

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 harddrive. 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.

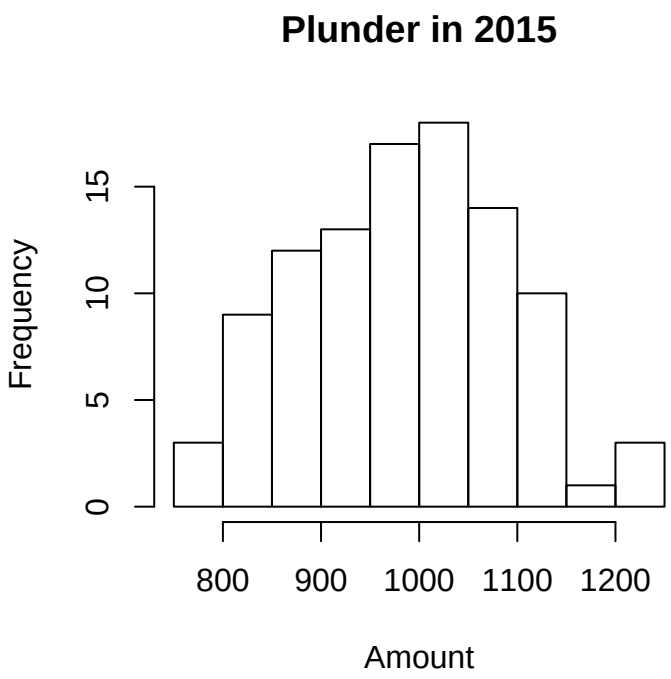
2. Plots - The Plots panel (no big surprise), shows all your plots. There are buttons for opening the plot in a separate window and exporting the plot as a pdf or jpeg (though you can also do this with code using the `pdf()` or `jpeg()` functions.)

   Let's see how plots are displayed in the Plots panel. Run the following code to display a histogram of 100 values randomly drawn from a standard normal distribution. When you do, you

Most - if not all - of the time when you perform actions using your mouse by pointing and clicking in RStudio, RStudio will perform the function by sending the appropriate R Code to the console. You can then copy and paste this code into your documents to automate the process later.

should see a plot similar to this one show up in the Plots panel. Don't worry if your plot looks slightly different from this one: as you'll learn later, the rnorm() function generates different data each time you evaluate it!

```
hist(x = rnorm(n = 100, mean = 1000, sd = 100),
     main = "Plunder in 2015",
     xlab = "Amount"
     )
```



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`

   To get help and see documentation for a function, type `?fun`, where `fun` is the name of the function. For example, to get additional information on the histogram function, run the following code:

Tip: If you ever need to learn more about an R function: type `?functionname`, where `functionname` is the name of the function.

```
?hist
```

Installing and loading packages

When you download and install R for the first time, you are installing the Base R software. Base R will will contain most if not all the functions you need. However, one of the great things about R is that people are constantly writing and sharing new functions that you can use. When people share a new function, they usually do so in the form of an *R package* which contains anything from functions, to help menus, to vignettes (examples), to data.

Most R packages are hosted at the Comprehensive R Archive Network (CRAN) https://cran.r-project.org/. To install a new R package from CRAN, you can simply run the code `install.packages("package")`, where "package" is the name of the package. After you've installed the package, you need to *load* it into R by running the code `library("package")`. This will load the package into your current R session and allow you to use its contents.

Once you've installed a package on your computer, you never need to install it again. However, you do need to load the package every time you start a new R session.

To see how this works in action, let's install and load the `wordcloud` package. This package contains the `wordcloud` function which allows you to easily create those really cool wordcloud plots you've seen on the Internets.

We'll start by installing the package. When you run the following code, 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.

```
install.packages("wordcloud")
```

Now that the package is installed on your computer, you can use it anytime you want by loading the package:

```
library("wordcloud")

## Loading required package: RColorBrewer
```

Now, let's create a wordcloud of pirate words! 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 wordcloud!

```
wordcloud(words = c("Blackbeard", "Jolly.Roger", "Grogg", "Monkey.Island", "Treasure", "cutlass"),
              freq = c(100, 20, 50, 40, 200, 100))
```

The yarrr package

For much of this book, you will need the `yarrr` package. This package contains every dataset, function, and plotting code from this book. Unlike most packages that you'll be using, the `yarrr` package is not hosted at CRAN. Instead, all the code is on github[2] at www.github.com/ndphillips/yarrr. To install the `yarrr` package on your machine, you'll first need to install the `devtools` package which will then allow you to directly install packages from github:

[2] For those of you unfamiliar with github, it's basically Facebook for programmers.

```
install.packages("devtools") # Install the devtools package
```

Now that you've installed and loaded the devtools package, you can install and load the `yarrr` package from github with the following code:

```
devtools::install_github("ndphillips/yarrr") # Install the yarrr package
```

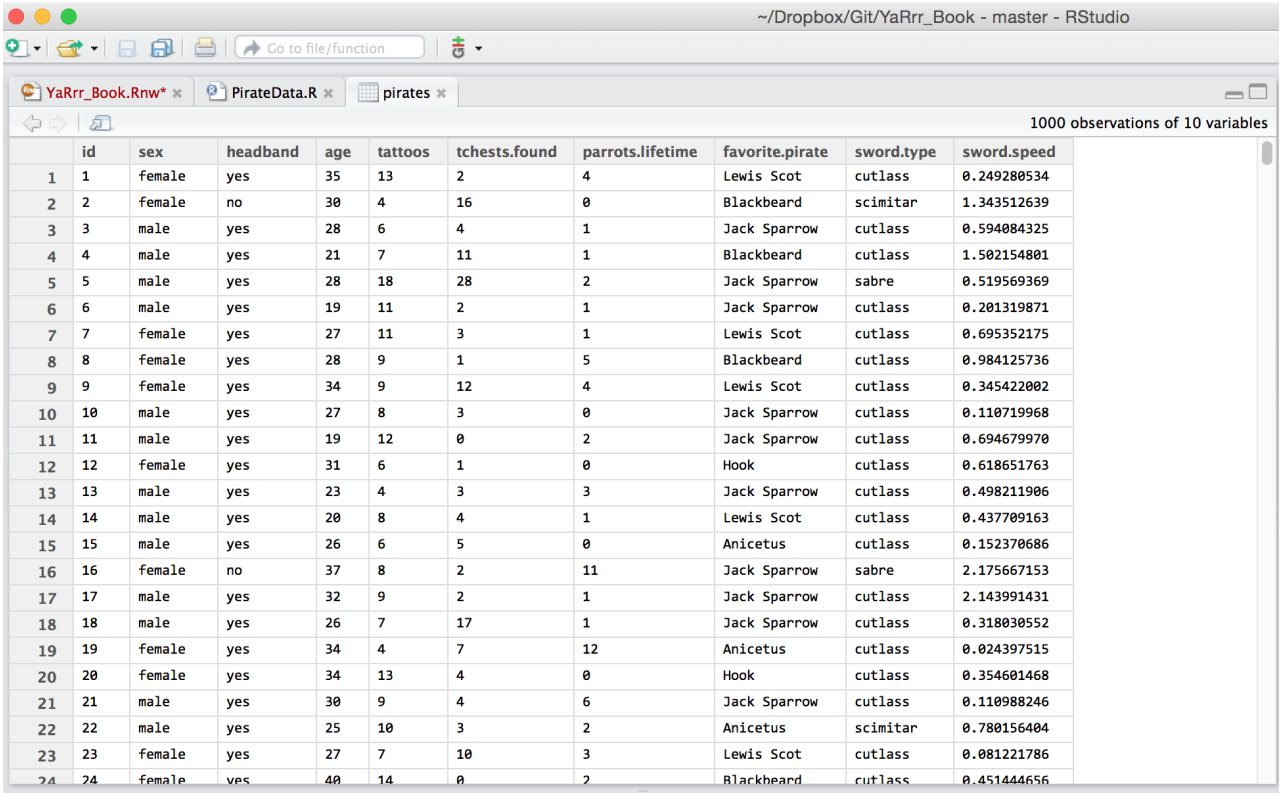
If everything went correctly, you should have access to all the datasets mentioned in this book in addition to many functions like `apa()` and `pirateplot()`.

Don't worry if you are unable to install the yarrr package. I'll also provide direct html links that you can use the download the datasets when necessary.

To see the dataset, you can execute the the `View()` function

```
View(pirates)
```

When you run this command, you should see the first several rows and columns of the dataset (like this:)



	id	sex	headband	age	tattoos	tchests.found	parrots.lifetime	favorite.pirate	sword.type	sword.speed
1	1	female	yes	35	13	2	4	Lewis Scot	cutlass	0.249280534
2	2	female	no	30	4	16	0	Blackbeard	scimitar	1.343512639
3	3	male	yes	28	6	4	1	Jack Sparrow	cutlass	0.594084325
4	4	male	yes	21	7	11	1	Blackbeard	cutlass	1.502154801
5	5	male	yes	28	18	28	2	Jack Sparrow	sabre	0.519569369
6	6	male	yes	19	11	2	1	Jack Sparrow	cutlass	0.201319871
7	7	female	yes	27	11	3	1	Lewis Scot	cutlass	0.695352175
8	8	female	yes	28	9	1	5	Blackbeard	cutlass	0.984125736
9	9	female	yes	34	9	12	4	Lewis Scot	cutlass	0.345422002
10	10	male	yes	27	8	3	0	Jack Sparrow	cutlass	0.110719968
11	11	male	yes	19	12	0	2	Jack Sparrow	cutlass	0.694679970
12	12	female	yes	31	6	1	0	Hook	cutlass	0.618651763
13	13	male	yes	23	4	3	3	Jack Sparrow	cutlass	0.498211906
14	14	male	yes	20	8	4	1	Lewis Scot	cutlass	0.437709163
15	15	male	yes	26	6	5	0	Anicetus	cutlass	0.152370686
16	16	female	no	37	8	2	11	Jack Sparrow	sabre	2.175667153
17	17	male	yes	32	9	2	1	Jack Sparrow	cutlass	2.143991431
18	18	male	yes	26	7	17	1	Jack Sparrow	cutlass	0.318030552
19	19	female	yes	34	4	7	12	Anicetus	cutlass	0.024397515
20	20	female	yes	34	13	4	0	Hook	cutlass	0.354601468
21	21	male	yes	30	9	4	6	Jack Sparrow	cutlass	0.110988246
22	22	male	yes	25	10	3	2	Anicetus	scimitar	0.780156404
23	23	female	yes	27	7	10	3	Lewis Scot	cutlass	0.081221786
24	24	female	yes	40	14	0	2	Blackbeard	cutlass	0.451444656

Figure 7: The pirates dataset included in the yarrr package.

The R Reference Card

Over the course of this book, you will be learning *lots* of new functions. Wouldn't it be nice if someone created a Cheatsheet / Notecard of many common R functions? Yes it would, and thankfully Tom Short has done this in his creation of the R Reference Card. I highly encourage you to print this out and start highlighting functions as you learn them! .

You can access the pdf of this card in two ways. You can get it on the web at http://nathanieldphillips.com/wp-content/uploads/2015/09/R-Reference-Card.pdf. Alternatively, you can get it from the yarrr package. To find where the file is located on your computer, run the

following code (the output after the code is the file location on my computer):

```
system.file("RReferenceCard.pdf", package="yarrr")

## [1] "/Library/Frameworks/R.framework/Versions/3.2/Resources/library/yarrr/RReferenceCard.pdf"
```

Finished!

That's it for this lecture! All you did was install the most powerful statistical package on the planet used by top universities and companies like Google. No big deal.

1.5: Jump off the plank and dive in

Figure 8: Despite what you might find at family friendly waterparks – this is NOT how real pirate swimming lessons look.

What's the best way to learn how to swim?

What's the first exercise on the first day of pirate swimming lessons? While it would be cute if they all had little inflatible 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!

Movies

In this section, we'll analyze a dataset of the top 5,000 grossing movies of all time. This dataset is called `movies` and is contained in the yarrr package.

First, we'll load the yarrr package. This will give us access to the movies dataset.

If you haven't downloaded the yarrr package yet, run the following commands:

```
install.packages('devtools')
library('devtools")
install_github('ndphillips/yarrr')
```

```
library(yarrr)
```

Next, we'll look at the help menu for the movies dataset using the question mark ?

```
?movies
```

First, let's take a look at the first few rows of the dataset using the `head()` function. This will give you a visual idea of how the dataset is structured.

```
head(movies)
```

Now let's calculate some basic statistics on the entire dataset. We'll calculate the mean revenue, highest running time, and number of movies in each genre[3].

[3] I got a mean revenue of 98.21, a maximum running time of 240, and a table which showed, among other things, that there were 692 Action movies, 486 Adventure movies...

```
# What is the mean boxoffice total of all movies?
mean(movies$revenue.all)

# What was the longest movie time?
max(movies$time, na.rm = T)

# How many movies are there of each genre?
table(movies$genre)
```

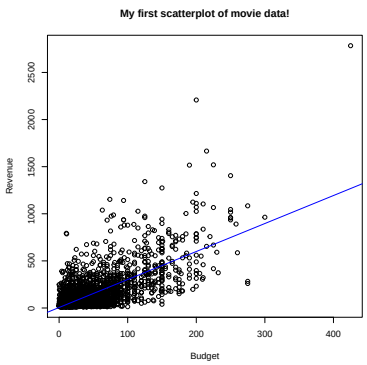
Now, let's calculate statistics for different gropus of data. For example, do you think that the budget of sequels is different from that of non-sequels? Let's calculate the median budget separately for movies that are sequels and those that are not[4].

[4] For non-sequels (where sequel = 0), I got a median budget of 28 Million. For sequels, I got a median budget of 48 Million.

```
aggregate(formula = budget ~ sequel,
          data = movies,
          FUN = median
          )
```

Cool stuff, now let's make a plot! We'll plot the relationship between movie budgets and revenue. I would hope that the higher a movie's budget is, the more money it should make. Let's find out if this is the case. We'll also include a blue regression line to measure the strength of the relationship.

Here's how my plot looks




```
plot(x = movies$budget,
     y = movies$revenue.all,
     main = 'My first scatterplot of movie data!',
     xlab = 'Budget',
     ylab = 'Revenue'
     )

abline(lm(revenue.all ~ budget,
          data = movies),
       col = 'blue'
       )
```

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 budget of sequels and non sequels[5]:

[5] I got a test statistic of -9.44 and a p-value of $2.1487864 \times 10^{-19}$ (which is pretty much 0).

```
t.test(formula = budget ~ sequel,
       data = movies,
       alternative = 'two.sided'
       )
```

Next, let's test if there a significant correlation between a movie's budget and its length. In other words, do movies with higher budgets tend to be longer?[6]

[6] For this correlation test, I got a correlation of 0.27, a test statistic of 11.65, and another p-value of $3.0811405 \times 10^{-30}$.

```
cor.test(formula = ~ budget + time,
         data = movies
         )
```

Finally, let's run a regression analysis to see if a movie's budget, length, and whether or not it is a sequel each predict how much a movie will make:

```
revenue.model <- lm(formula = revenue.all ~ budget + time + sequel,
                    data = movies
                    )
summary(revenue.model)
```

Now, let's repeat some of our previous analyses with Bayesian versions. First we'll install[7] and load the `BayesFactor` package which contains the Bayesian statistics functions we'll use:

[7] You need to be connected to the internet in order to install the BayesFactor package. If you aren't, you'll have to skip the rest of these exercises.

```
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 the t-test comparing the budgets of movie sequels to non-sequels[8] :

[8] I got a Bayes Factor of 5.0869775×10^{38} which is pretty darn strong evidence against the null hypothesis.

```
ttestBF(formula = budget ~ sequel,
        data = movies[is.finite(movies$budget) &
                      is.finite(movies$sequel),])
```

Now, let's test the correlation between a movie's budget and its revenue with a Bayesian regression analysis[9]:

[9] I got a Bayes Factor of ∞ (Infinite!) which is such strong evidence against the null hypothesis that the time and space continuum is on the verge of collapsing.

```
lmBF(formula = revenue.all ~ budget,
        data = movies[is.finite(movies$revenue.all) &
                      is.finite(movies$budget),])
```

Wasn't that easy?!

Ok seriously. Wasn't that easy? Imagine how long it would have taken to explain how to do all that in SPSS[10]. 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,

[10] Oh wait I forgot, SPSS can't even do Bayesian statistics

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 gotten wet, let's learn how to swim.

2: R Basics

Figure 9: 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.

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.

The basics of R programming

Objects and functions. Functions and objects

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.

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 *scalers* which represent single numbers, *vectors* which represent a list of 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 takes a vector of numeric data as an argument, calculates

the arithmetic mean of those data, then returns a single number (a scaler) 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.

Let's see a built-in function in practice. In the code below, I'll define a vector object called *tattoos*. I'll then use the *mean()* function to calculate the average number of tattoos in the tattoos vector by setting the argument as the tattoos vector.

```
# 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] 23.83333
```

As you can see, the result of the `mean()` function, with the tattoos vector object as an input, returned a single number (a scaler object) with a value of 23.8333333.

Different functions require different types of objects as inputs. For example, the `mean()` function has to have a *numeric* object (typically a vector) as an input. If you try to use a non-numeric object as an argument to `mean()`, you'll get an error because the function wasn't designed to work with that type of argument.

As you learn R, you'll spend a lot of time learning new types of objects (and their attributes) and new functions. To see a list of the most commonly used built-in functions and procedures in R, check out the great R Reference Card written by Tom Short at http://nathanieldphillips.com/wp-content/uploads/2015/09/R-Reference-Card.pdf. I *highly* encourage you to print the R Reference Card out and keep it handy while you're learning new functions.

The command-line interpreter

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
```

As you can see, R returned the (thankfully correct) value of 2[11]. 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 idiot. It (and they), are much more than that.

Figure 10: 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.

[11] 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.

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[12] in an editor.

In RStudio, you'll write your R code in the...wait for it...*Editor* window. To start writing a new R script in RStudio, click File – New File – R Script[13]. 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 11, I have a new script called "myfirstscript.R" with a few random calculations.

[12] 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.we

[13] **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.

Send code from an editor 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 12).

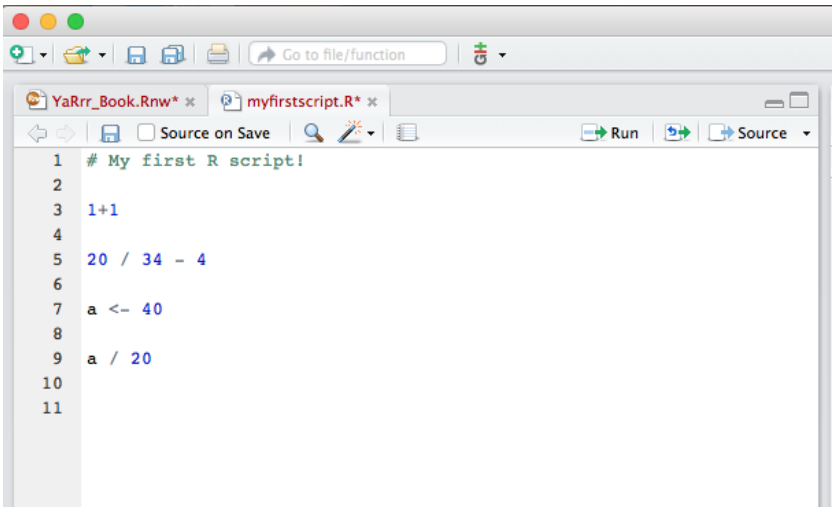


Figure 11: Here's how a new script looks in the editor window on RStudio. The code you type won't be executed until you send it to the console.

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.

Figure 12: 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

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[14].

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:

[14] 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

Figure 13: 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.

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 Strager 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.

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:

```
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

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.

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[15] called "assign."

```
object <- [ ]
```

[15] You can use = instead of <- for object assignment but I recommend you stick with <- because the direction of the assignment is clear.

To assign something to a new object (or to update 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 carpel tunnel syndrome.

```
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:

```
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 in eval(expr, envir, enclos): 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
```

Top 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
```

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.

How to name objects

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

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 syntax error of some kind.

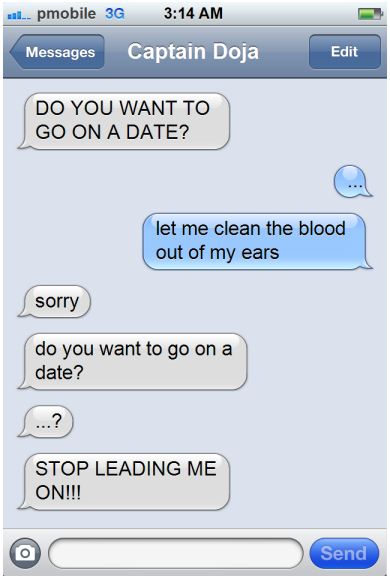


Figure 14: Like a text message, you should probably watch your use of capitalization in R.

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
```

Avoid using too many capital letters in object names because they require you 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.

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] 558759611
```

It looks like the movie made 5.5875961×10^8 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] 1.679201
```

It looks like "Dead Man's Chest" made 68% more than "Curse of the Black Pearl" - not bad for two movies based off of a ride from Disneyland.

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
this.one <- 3
```

```
This.1 <- 4
ThIS.....ON...E <- 5
This!One! <- 6
lkjasdfkjsdf <- 7
```

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
```

3: Scaler and vector objects

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

Scalers

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

```
a <- 100
b <- 3 / 100
c <- (a + b) / c
```

Scalers 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 scalers:

```
d <- "ship"
e <- "cannon"
f <- "Do any modern armies still use cannons?"
```

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

```
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.

```
# scalar v vector v matrix

par(mar = rep(1, 4))
plot(1, xlim = c(0, 4), ylim = c(-.5, 5),
     xlab = "", ylab = "",
     xaxt = "n", yaxt = "n",
     bty = "n", type = "n")

# scalar
rect(rep(0, 1), rep(0, 1), rep(1, 1), rep(1, 1))
text(.5, -.5, "scalar")

# Vector
rect(rep(2, 5), 0:4, rep(3, 5), 1:5)
text(2.5, -.5, "Vector")
```

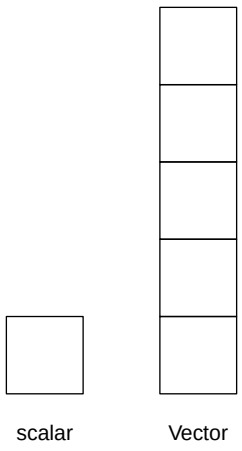


Figure 15: Visual depiction of a scalar and vector. Deep shit. Wait until we get to matrices - you're going to lose it.

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:

Function	Example	Result
`c(a, b)`	`c(1, 5, 9)`	[1, 5, 9]
`a:b`	`5:10`	[5, 6, 7, 8, 9, 10]
`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(1, 5), times = 2, each = 2)`	[1, 1, 5, 5, 1, 1, 5, 5]

Ok now it's time to learn our first function! We'll start with `c()` which allows you to build vectors.

`c(a, b, c, ...)`

`a, b, c, ...`

One or more objects to be combined into a vector

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 scalers as arguments, and returns a vector containing those objects. When using c(), place a comma in between the objects (scalers 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.

```
a <- c(1, 2, 3, 4, 5)
```

Let's look at the object by evaluating it in the console:

```
a

## [1] 1 2 3 4 5
```

As you can see, R has stored all 5 numbers in the object `a`. Thanks R!

You can also create longer vectors by combining vectors you have already defined. Let's create a vector of the numbers from 1 to 10 by first generating a vector `a` from 1 to 5, and a vector `b` from 6 to 10 then combine them into a single vector `c`:

```
a <- c(1, 2, 3, 4, 5)
b <- c(6, 7, 8, 9, 10)
c <- c(a, b)
c

##  [1]  1  2  3  4  5  6  7  8  9 10
```

You can also create character vectors by using the `c()` function to combine character scalers into character vectors:

```
char.vec <- c("this", "is", "not", "a", "pipe")
char.vec

## [1] "this" "is"   "not"  "a"    "pipe"
```

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 scalers.

```
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.

Functions to generate numeric vectors

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()`:

a:b

The `a:b` function takes two scalers `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. You can go forwards or backwards depending on which number is larger.

a:b

`a`

The start of the sequence

`b`

The end of the sequence

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
```

seq()

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:

seq(from, to, by)

`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 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:

```
seq(from = 1, to = 10, by = 1)

##  [1]  1  2  3  4  5  6  7  8  9 10

seq(from = 0, to = 100, by = 10)

##  [1]   0  10  20  30  40  50  60  70  80  90 100
```

If you use the `length.out` argument, the sequence will have length equal to the input of `length.out`.

```
seq(from = 0, to = 100, length.out = 11)

##  [1]   0  10  20  30  40  50  60  70  80  90 100

seq(from = 0, to = 100, length.out = 5)

## [1]   0  25  50  75 100
```

rep()

The `rep()` function allows you to repeat a scaler (or vector) a specified number of times.

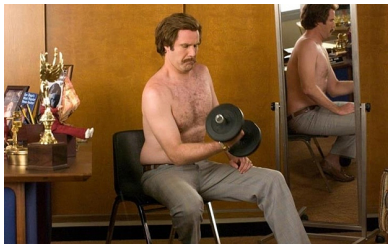


Figure 16: Not a good depiction of a rep in R.

rep(x, times, each)

`x`

A scaler or vector of values to repeat

`times`

The number of times to repeat the sequence

`each`

The number of times to repeat each value within the sequence

`length.out (optional)`

The desired length of the final sequence

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), times = 5)

##  [1] 1 2 1 2 1 2 1 2 1 2

rep(x = c("a", "b"), each = 2)

## [1] "a" "a" "b" "b"

rep(x = 1:3, length.out = 10)

##  [1] 1 2 3 1 2 3 1 2 3 1

rep(x = 1:5, times = 3)

##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

As you can see in the fourth example above, 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
```

Assigning jobs to a new crew

Let's say you are are getting a batch of 10 new pirates on your crew, and you need to assign each of them to one of three jobs. To help you, you could use a vector with the numbers 1, 2, 3, 1, 2, 3, etc. As they board the ship, you'll just read off the next job to each pirate. Let's create this vector using `rep()`

```
pirate.jobs.num <- rep(x = 1:3,
                       length.out = 10)

pirate.jobs.num

##  [1] 1 2 3 1 2 3 1 2 3 1
```

Ok that's helpful, but it would be nice if the jobs were in text instead of numbers. Let's repeat the previous example, but instead of using the numbers 1, 2, 3, we'll use the names of the actual jobs (which are Deck Swabber, Parrot Groomer, and App Developer)

```
pirate.jobs.char <- rep(x = c("Deck Swabber", "Parrot Groomer", "App Developer"),
                        length.out = 10)
pirate.jobs.char

##  [1] "Deck Swabber"   "Parrot Groomer" "App Developer"  "Deck Swabber"
##  [5] "Parrot Groomer" "App Developer"  "Deck Swabber"   "Parrot Groomer"
##  [9] "App Developer"  "Deck Swabber"
```

The each argument allows you to repeat each element in the original vector within each repetition. Wow, that was a confusing sentence. Let me just show you:

```
rep(x = 1:4, each = 2)

## [1] 1 1 2 2 3 3 4 4

rep(x = c("a", "b"), each = 3, times = 2)

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

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 an object called `overboard` containing the text "NNNNNNOOOOOOoooooo!!!!!!" as a vector of length 24 using the `rep()` function.

4: Vector functions

In this chapter, we'll cover some basic vector operations and functions.

length()

Once you have a vector, you may want to know how long it is. When this happens, don't stare at your computer screen and count the elements one by one! Instead, use `length()` function. The `length()` function takes a vector as an argument, and returns a scaler representing the number of elements in the vector:

```
a <- 1:10
length(a)

## [1] 10

b <- seq(from = 1, to = 100, length.out = 20)
length(b)

## [1] 20

length(c("This", "character", "vector", "has", "six", "elements."))

## [1] 6

length("This character scaler has just one element.")

## [1] 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.

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 scaler objects. Let's create a vector with the integers from 1 to 10, and add then add 100 to each element:

```
a <- 1:10
a + 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 `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 scaler, R will just apply the operation to every element in the vector.

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
```

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
```

Summary statistic functions for numeric vectors

Ok, now that we can create vectors, let's learn the basic descriptive statistics functions. We'll start with functions that apply to both continuous and discrete data[16]. Each of the following functions takes a numeric vector as an argument, and returns either a scalar (or in the case of `summary()`, a `table`) as a result.

[16] Continuous data is data that, generally speaking, can take on an infinite number of values. Height and weight are good examples of continuous data. Discrete data are those that can only take on a finite number of values. The number of pirates on a ship, or the number of times a monkey farts in a day are great examples of discrete data

Statistical functions for numeric vectors

`sum(x), product(x)`

The sum, or product, of a numeric vector x.

`min(x), max(x)`

The minimum and maximum values of a vector x

`mean(x)`

The arithmetic mean of a numeric vector x

`median(x)`

The median of a numeric vector x. 50% of the data should be less than `median(x)` and 50% should be greater than `median(x)`.

`sd(x), var(x)`

The standard deviation and variance of a numeric vector x.

`quantile(x, p)`

The pth sample quantile of a numeric vector x. For example, `quantile(x, .2)` will tell you the value at which 20% of cases are less than x. The function `quantile(x, .5)` is identical to `median(x)`

`summary(x)`

Shows you several descriptive statistics of a vector x, including `min(x)`, `max(x)`, `median(x)`, `mean(x)`

Let's calculate some descriptive statistics from some pirate related data. I'll create a vector called `data` 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

max(tattoos)

## [1] 100

mean(tattoos)
```

```
## [1] 24.1

median(tattoos)

## [1] 9

sd(tattoos)

## [1] 31.32074
```

Watch out for 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:

Important!!! Include the argument na.rm = T to ignore missing (NA) values when calculating a descriptive statistic.

```
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 = T` 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 = T` argument:

```
mean(a, na.rm = T)

## [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.

If you want to get many summary statistics from a vector, you can use the **summary()** function which gives you several key statistics:

```
summary(tattoos)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.00    4.00    9.00   24.10   34.25  100.00
```

Additional numeric vector functions

Here are some other functions that you will find useful when managing numeric vectors:

Additional numeric vector functions

`round(x, digits)`

Round values in a vector (or scaler) x to a certain number of digits.

`ceiling(x), floor(x)`

Round a number to the next largest integer with `ceiling(x)` or down to the next lowest integer with `floor(x)`.

`x %% y`

Modular arithmetic (i.e.; x mod y). You can interpret `x %% y` as "What is the remainder after dividing x by y?" For example, `10 %% 3` equals 1 because 3 times 3 is 9 (which leaves a remainder of 1).

Counting functions for discrete and non-numeric data

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

Counting functions for discrete data

`unique(x)`

Returns a vector of all unique values in the vector x.

`table(x)`

Returns a table showing all the unique values in the vector x as well as a count of each occurrence. By default, the `table()` function does NOT count NA values. To include a count of NA values, include the argument `exclude = NULL`

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 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
## 0.5555556 0.1111111 0.3333333

table(gender) / sum(table(gender))

## gender
##         F         M
## 0.5555556 0.4444444
```

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:

```
a <- c(5, 3, 7, 5, 5, 3, 4)
mean(a)

## [1] 4.571429

sd(a)

## [1] 1.397276
```

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)
```

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] 1.982309e-16

sd(a.z)

## [1] 1
```

Sweet[17].

[17] Oh, don't worry that the mean of `a.z` doesn't look like exactly zero. Using non-scientific notation, the result is 0.00000000000000198. 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)

A worked example: 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:

```
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.

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. To make them easier to read, I'll round them to 2 digits:

```
round(grogg.z, 2)

## [1]  1.38  0.49 -1.07  0.04 -0.85

round(climbing.z, 2)

## [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 had a z-score of 1.4. We can interpret this by saying that this pirate drank 1.4 more standard deviations of mugs of grogg than the average pirate. In the climbing competition, the fifth pirate had a z-score of 1.3. Here, we would conclude that this pirate 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 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.

Test your R Might!

1. Create a vector that shows the square root of the integers from 1 to 10.

2. 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 1000 bushels 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[18].

3. 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, and then did the same over 7 days while drinking grogg. Here are her results:

```
grogg <- c(2, 0, 3, 1, 0, 3, 5)
nogrogg <- c(0, 0, 1, 0, 1, 2, 2)
```

   How much treasure did Renata find on average when drinking grogg? What about when she did not drink grogg?

[18] 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.

4. Using Renata's data again, create a new vector called `difference` that shows how much more treasure Renata found while drinking grogg than when she didn't drink grogg. What was the mean, median, and standard deviation of the difference?

5: Working with vectors

By now you should feel comfortable applying functions like `mean()` and `table()` to vectors. However, in many analyses, you will want to access specific values of a vector based on their position or on some other criteria. For example, you may want to analyze values in a specific location in the vector (i.e.; the first 10 elements) or based on some criteria (i.e.; all values greater than 0). To access specific values of a vector in R, we use *indexing*.

To show you where we're going with this, consider the following two vectors `sex` and `beard` – these vectors represent responses to two questions in a survey of five pirates. `sex` is a character vector representing the pirates' sex (m or f), while beard is a numeric vector indicating the length of the pirates' beards.

```
sex <- c("m", "m", "f", "m", "f")
beard <- c(30, 24, 0, 40, 0)
```

Here's how we can use indexing to answer questions about subsets of the data:

```
# What is the length of the first pirate's beard?
beard[1]

## [1] 30

# What is the sex of the 3rd, 4th, and 5th pirates?
sex[3:5]

## [1] "f" "m" "f"

# How many females were there?
sum(sex == "f")

## [1] 2

# What are the values of beard that are greater than 0?
beard[beard > 0]

## [1] 30 24 40
```

Figure 17: Traditionally, a pirate will steal a Beard Bauble from every pirate he beats in a game of Mario Kart. This pirate is is the Mario Kart champion of Missionsstrasse.

```
# What are the beard lengths of males?
beard[sex == "m"]

## [1] 30 24 40
```

As you can see, we use brackets [ ] in *index* (aka, select) specific values of a vector. In this chapter we'll cover these and other ways of indexing vectors.

Indexing vectors with brackets

a[]

To index a vector, we use brackets [ ] after the vector object. 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.

Numerical Indexing

With numerical indexing, you enter the 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, to get the first value in a vector called a, you'd write `a[1]`. To get the first, second, and third value, you can either type `a[c(1, 2, 3)]` or `a[1:3]`. As you can see, you can use any of the vector generation functions we learned in the previous chapter to index a vector. However, the values must be integers.

Let's do a few more examples. We'll create an a vector again and use indexing to extract specific values:

```
a <- c(0, 50, 2, 39, 9, 20, 17, 8, 10, 100)

# 1st element
a[1]

## [1] 0

# 1-5 elements
a[1:5]

## [1]  0 50  2 39  9

# Every second element
a[seq(from = 2, to = 10, by = 2)]

## [1]  50  39  20   8 100
```

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

```
my.index <- 6:10
a[my.index]

## [1]  3  4 NA NA NA
```

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 either TRUE or T, and false values with either FALSE or F.

You can 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 19. For example, let's start with a vector a containing five numbers:

```
a <- c(5, 3, -2, -5, 11)
```

We can create logical vectors by comparing the original vector a to a scaler as follows:

```
# Which values are > 0?
a > 0

## [1]  TRUE  TRUE FALSE FALSE  TRUE

# Which values are equal to 5?
a == 5

## [1]  TRUE FALSE FALSE FALSE FALSE
```

For character vectors, you can also use the == (equals) or != (not-equal) comparisons (other comparisons such as < don't make sense for character vectors).

```
b <- c("m", "f", "f", "f", "m")

b == "m"

## [1]  TRUE FALSE FALSE FALSE  TRUE

b != "m"

## [1] FALSE  TRUE  TRUE  TRUE FALSE
```

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.)

Figure 18: Logical indexing. Good for R aliens and R pirates.

```
par(mar = rep(.1, 4))
plot(1, xlim = c(0, 1.1), ylim = c(0, 9),
     xlab = "", ylab = "", xaxt = "n", yaxt = "n",
     type = "n")


text(rep(0, 8), 8:1,
     labels = c("==", "!=", "<", "<=",
                ">", ">=", "|", "!"),
     adj = 0, cex = 3)

text(rep(.2, 8), 8:1,
     labels = c("equal", "not equal", "less than",
                "less than or equal","greater than",
                "greater than or equal", "or", "not"),
        adj = 0, cex = 3)
```

==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
|	or
!	not

Figure 19: Comparison operators in R

```
a <- c(1, 2, 3, 4, 5)
b <- c(3, 3, 3, 3, 3)

a > b

## [1] FALSE FALSE FALSE  TRUE  TRUE

a != b

## [1]  TRUE  TRUE FALSE  TRUE  TRUE
```

Once you create a logical vector, you can use that logical vector to index another vector. When you do this, R will return the values of the vector for all TRUE values of the logical vector, and will ignore all values for which the logical vector is FALSE. 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.

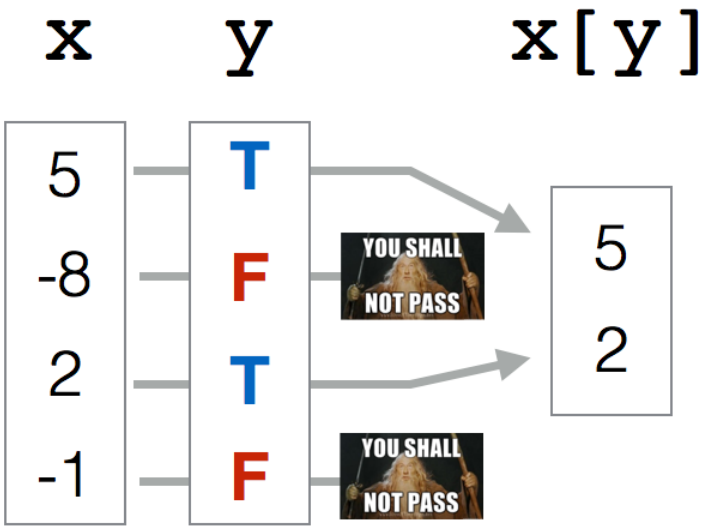


Figure 20: FALSE values in a logical vector are like lots of mini-Gandolfs. In this example, I am indexing a vector x with a logical vector y (y for example could be x > 0, so all positive values of x are TRUE and all negative values are FALSE). The result is a vector of length 2, which are the values of x for which the logical vector y was true. Gandolf stopped all the values of x for which y was FALSE.

For example, let's look at our sex and beard data again.

```
sex

## [1] "m" "m" "f" "m" "f"

beard

## [1] 30 24  0 40  0
```

Now, we'll use the logical vector (`sex == "f"`) to index the `beard` vector. This will give us the values of `beard` (representing beard lengths) *only* for cases where the `sex` vector is equal to f:

```
beard[sex == "f"]

## [1] 0 0
```

This result tells us there were two females (that is, there were two TRUE values in the logical vector `sex == "f"`), and both had a beard length of 0.

Using & (AND), | (OR)

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, in addition to our `sex` and `beard` vectors, let's create an age vector indicating the age of the 5 pirates:

```
age <- c(19, 24, 22, 40, 88)
```

Now, we can use the `&` (AND), and `|` (OR) operations to create a logical vector based on both age and sex criteria. In this example, I'll create a logical vector for females (`sex == "f"`) who are older than 20 (`age > 20`):

```
sex == "f" & age > 20

## [1] FALSE FALSE  TRUE FALSE  TRUE
```

Because the result is just another logical vector, we can use it to index our main data vector (`beard`).

```
# Beard lengths of female AND older than 20?
beard[sex == "f" & age > 20]

## [1] 0 0

# Beard lengths of male OR less than 40?
beard[sex == "m" | age < 40]

## [1] 30 24  0 40
```

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 for females with an age less than 20 OR greater than 35, and whose beard length is not equal to 20

```
sex == "f" & (age < 20 | age > 35) & beard != 20

## [1] FALSE FALSE FALSE FALSE  TRUE
```

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

Additional ways to create and use logical vectors

`x %in% y`

%in%

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.

```
x %in% c("a", "b", "c", "d")

## [1]  TRUE FALSE  TRUE  TRUE FALSE
```

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.

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

Changing values in a vector with indexing

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  6  7  8  9 10
```

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 the integers from 1 to 10:

```
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  6  7  8  9 10
```

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  6  7  8  9 10
```

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.

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
```

Using indexing to change specific values of a vector

You can also change values of a vector with logical indexing. This 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. If we try to take the mean of this vector, we'll get a nonsense response:

```
mean(happy)

## [1] 201.3
```

We need to somehow take care of these invalid values. To do this, we'll use logical indexing to change the invalid values (999 and -2) to NA. We'll start by creating a vector of valid responses:

```
valid.response <- 1:5
```

Now, 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).

```
log.vec <- (happy %in% valid.response) == FALSE
log.vec

##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

Finally, we'll index `happy` with `log.vec`, and assign the invalid values as NA:

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

```
happy[(happy %in% 1:5) == FALSE] <- NA
```

```
happy[log.vec] <- NA
happy

##  [1]  1  4  2 NA  2  3 NA  3  2 NA
```

As you can see, the new vector `happy.valid` replaced the invalid values with NAs. Now we can take a `mean()` of the vector and see the mean of the valid responses.

```
mean(happy, na.rm = T)

## [1] 2.428571
```

Of course, because we now have NA values in our vector, we need to include the `na.rm = T` argument to the `mean()` function! If we don't, we'll just get an NA response:

```
mean(happy)

## [1] NA

mean(happy, na.rm = T)

## [1] 2.428571
```

Taking the sum and mean of logical vectors to get counts and percentages

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

```
x <- c(5, 2, -2, -7, 2, -6, -2, 1, 8, 10)
```

Now, we'll use `sum()` and `mean()` to see how many of the values in x are positive, and what percent are positive. We should find that there are 6 TRUE values, and that 60% of the values (6 / 10) are TRUE.

```
sum(x > 0)

## [1] 6

mean(x > 0)

## [1] 0.6
```

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.

Test your R Might!: Movie data

The following vectors contain data about 10 of my favorite movies.

```
m.names <- 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?
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 $30 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 and were comedies?

Additional Tips

R has lots of special functions that take vectors as arguments, and return logical vectors based on multiple criteria. Here are some that I frequently use:

Logical testing functions

`is.integer(x)`

Tests if values in a vector are integers

`is.na(x), is.null(x)`

Tests if values in a vector are `NA` or `NULL`

`is.finite(x)`

Tests if a value is a finite numerical value. If a value is NA, NULL, Inf, or -Inf, `is.finite()` will return `FALSE`.

`duplicated(x))`

Returns `FALSE` at the first location of each unique value in x, and `TRUE` for all future locations of unique values. For example, `duplicated(c(1, 2, 1, 2, 3))` returns (FALSE, FALSE, TRUE, TRUE, FALSE). If you want to remove duplicated values from a vector, just run `x <- x[!duplicated(x)]`

which(log.vec)

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, given our `sex` vector, we can use the `which()` function to see which pirates are male and which are female:

```
which(sex == "m")

## [1] 1 2 4

which(sex == "f")

## [1] 3 5
```

6: Matrices and Data Frames

Figure 21: Did you actually think I could talk about matrices without a Matrix reference?!

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. See Figure 22 to see the shocking difference between these data objects.

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[19].

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.

```r
# scalar v vector v matrix

par(mar = rep(1, 4))
plot(1, xlim = c(0, 10), ylim = c(-.5, 5),
     xlab = "", ylab = "",
     xaxt = "n", yaxt = "n",
     bty = "n", type = "n")

# scalar
rect(rep(0, 1), rep(0, 1), rep(1, 1), rep(1, 1))
text(.5, -.5, "scalar")

# Vector
rect(rep(2, 5), 0:4, rep(3, 5), 1:5)
text(2.5, -.5, "Vector")

# Matrix
rect(rep(4:8, each = 5),
     rep(0:4, times = 5),
     rep(5:9, each = 5),
     rep(1:5, times = 5))
text(6.5, -.5, "Matrix / Data Frame"
     )
```

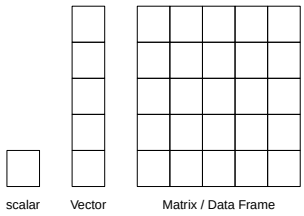


Figure 22: scalar, Vector, Matrix... ::drops mike::

[19] **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.

Creating matrices and dataframe objects

There are a number of ways to create your own matrix and dataframe objects in R. 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.

Creating matrices and dataframes

Function	Description
`cbind()`	Combine vectors as *columns* in a matrix/dataframe
`rbind()`	Combine vectors as *rows* in a matrix/dataframe
`matrix()`	Create a matrix with a desired number of rows and columns from a single vector
`data.frame()`	Combine vectors as columns in a dataframe

cbind() and 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 10, then we'll combine them into one matrix.

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

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

##      x  y  z
## [1,] 1  6 11
## [2,] 2  7 12
## [3,] 3  8 13
## [4,] 4  9 14
## [5,] 5 10 15

# Create a matrix where x, y and z are rows
rbind(x, y, z)
```

cbind()

rbind()

Keep in mind that matrices can either contain numbers or characters. If you try to create a matrix with both numbers and characters, it will turn all the numbers into characters:

```
cbind(1:5,
      c("a", "b", "c", "d", "e"))

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

```
##   [,1] [,2] [,3] [,4] [,5]
## x    1    2    3    4    5
## y    6    7    8    9   10
## z   11   12   13   14   15
```

As you can see, the `cbind()` function combined the vectors as columns in the final matrix, while the `rbind()` function combined them as rows.

matrix()

The `matrix()` function creates a matrix form a single vector of data. The function has 3 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
```

matrix()

We can also organize the vector by rows instead of columns using the argument `byrow = T`:

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

matrix(data = 1:10,
       nrow = 2,
       ncol = 5,
       byrow = T)

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
```

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.

`data.frame()`

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

```
# Create a dataframe with columns col.1,
#  col.2, and col.3

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

##   index sex age
## 1     1   m  99
## 2     2   m  46
## 3     3   m  23
## 4     4   f  54
## 5     5   f  23
```

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. Here are a few datasets that we will be using in future examples:

To see a complete list of all the datasets included in the `datasets` package, run the code: `library(help = "datasets")`

- `ChickWeight`: Weight versus age of chicks on four different diets
- `InsectSprays`: Effectiveness of six different types of insect sprays
- `ToothGrowth`: The effects of different levels of vitamin C on the tooth growth of guinea pigs.

Since these datasets are preloaded in R, you can always access them without having to create them manually. For example, if you run `head(ToothGrowth)`, you'll see the first few rows of the `ToothGrowth` dataframe.

Matrix and dataframe functions

R has lots of functions for viewing matrices and dataframes and returning information about them. Here are the most common:

Matrix and Dataframe Functions

Function	Description
`head()`	Print the *first* few rows to the console
`tail()`	Print the *last* few rows to the console
`View()`	Open the entire object in a new window
`dim()`	Count the number of rows and columns
`nrow()`, `ncol()`	Count the number of rows (or columns)
`rownames()`, colnames()	Show the row (or column) names
`names()`	Show the column names (only for dataframes)
`str()`	Show the structure of a dataframe
`summary()`	Show summary statistics

View()

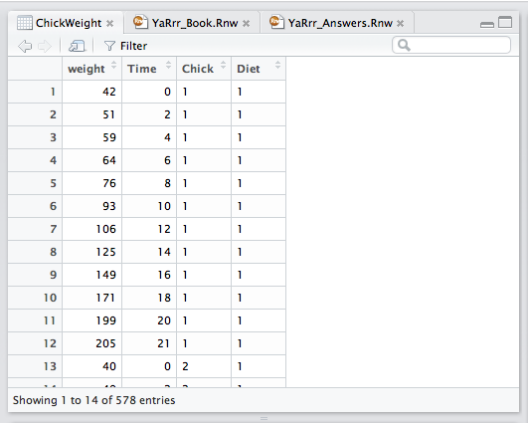
You can print an entire matrix or dataframe in the console by just executing the name of the object, but if the matrix or dataframe is very large, it can overwhelm the console. Instead, it's best to use one of the viewing functions like `View()` or `head()`. The `View()` function will open the object in its own window:

`View()`

```
# Print the entire ChickWeight dataframe
#  in a new data window

View(ChickWeight)
```

head()

To print the first few rows of a dataframe or matrix to the console, use the `head()` function:



	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

Figure 23: 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.

head()

```
# Print the first few rows of ChickWeight
#  to the console

head(ChickWeight)

##   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
```

summary()

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

summary()

```
# Print summary statistics of the columns of
#  ChickWeight to the console

summary(ChickWeight)

##      weight           Time           Chick     Diet
##  Min.   : 35.0   Min.   : 0.00   13     : 12   1:220
##  1st Qu.: 63.0   1st Qu.: 4.00   9      : 12   2:120
##  Median :103.0   Median :10.00   20     : 12   3:120
##  Mean   :121.8   Mean   :10.72   10     : 12   4:118
##  3rd Qu.:163.8   3rd Qu.:16.00   17     : 12
##  Max.   :373.0   Max.   :21.00   19     : 12
```

```
##                                   (Other):506
```

str()

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.

str()

```
# Print the classes of the columns of
#  ChickWeight to the console

str(ChickWeight)

## Classes 'nfnGroupedData', 'nfGroupedData', 'groupedData' and 'data.frame': 578 obs. of  4 variables:
##  $ weight: num  42 51 59 64 76 93 106 125 149 171 ...
##  $ Time  : num  0 2 4 6 8 10 12 14 16 18 ...
##  $ Chick : Ord.factor w/ 50 levels "18"<"16"<"15"<..: 15 15 15 15 15 15 15 15 15 15 ...
##  $ Diet  : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
##  - attr(*, "formula")=Class 'formula' length 3 weight ~ Time | Chick
##   .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
##  - attr(*, "outer")=Class 'formula' length 2 ~Diet
##   .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
##  - attr(*, "labels")=List of 2
##   ..$ x: chr "Time"
##   ..$ y: chr "Body weight"
##  - attr(*, "units")=List of 2
##   ..$ x: chr "(days)"
##   ..$ y: chr "(gm)"
```

Here are some of the other functions in action:

```
# What are the names of the ChickWeight columns?
names(ChickWeight)

## [1] "weight" "Time"   "Chick"  "Diet"

# How many rows are in ChickWeight?
nrow(ChickWeight)

## [1] 578

# How many columns are in ChickWeight?
ncol(ChickWeight)

## [1] 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.

names()

names()

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 the ToothGrowth dataframe?
names(ToothGrowth)

## [1] "len"  "supp" "dose"
```

Accessing dataframe columns by name with $

df$column

To access a specific column in a dataframe by name, you use the `$` operator in the form `df$colname` where `df` is the name of the dataframe, and `colname` 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` dataset:

```
# 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
```

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
ToothGrowth[c("len", "supp")]
```

Because the $ operator returns a vector, you can easily calculate descriptive statistics on columns of a dataframe 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] 18.81333

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

```
##
## OJ VC
## 30 30
```

Adding new columns to a dataframe

You can add new columns to a dataframe using the `$` and assignment `<-` operators. To do this, just use the `dataframe$colname` notation and assign a new vector to it. Let's test this by adding a new column to `ToothGrowth` called `len.in` which converts the original `len` column from centimeters to inches. Because 1 cm = 2.54 inches, we'll just multiply the original `len` column by 2.54.

```
df$new <- [ ]
```

```
# Add a new column called len.in to
#  ToothGrowth calculated as len * 2.54

ToothGrowth$len.in <- ToothGrowth$len * 2.54
```

You can add new columns with any information that you want to a dataframe - even basic numerical vectors. For example, let's add a simple `index` column to the dataframe showing the original order of the rows of the data. To do this, I'll assign the numeric vector `1:nrow(ToothGrowth)` to a new column called `index`

```
# Add a new columnn called index to
#  ToothGrowth with integers 1:nrow(ToothGrowth)

ToothGrowth$index <- 1:nrow(ToothGrowth)
```

Changing dataframe column names

To change the name of a column in a dataframe, just use a combination of the `names()` function, indexing, and reassignment. For example, to change the name of the first column of `ToothGrowth` to "length" just index the first value of `names(ToothGrowth)` and reassign:

```
# Change the name of the first column of
#  ToothGrowth to "length"

names(ToothGrowth)[1] <- "length"
```

Tip! If you have a dataframe with several columns, and you want to change one of the column names, you can avoid having to visually look up the index by utilizing the `which()` function. For example, in a dataframe called `df`, you can change a column titled "old" to "new' as follows:

```
# Change the name of the "old" column
#  in df to "new"
names(df)[which(names(df) == "old")] <- "new"
```

If you have a dataset with, say, over 100 variables, this trick can save you a lot of time and eye-strain.

Slicing and dicing 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 and dicing*.

Figure 24: Slicing and dicing data. The turnip represents your data, and the knife represents indexing with brackets, or subsetting functions like `subset()`. The black-eyed clown holding the knife is just off camera.

`df[rows, columns]`

Indexing dataframes with brackets [rows, columns]

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.

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 1-5 and column 1 of ToothGrowth
ToothGrowth[1:5, 1]

## [1]  4.2 11.5  7.3  5.8  6.4
```

Because the first column is `len`, the primary dependent measure, this means that the tooth lengths in the first 5 observations are 4.2, 11.5, 7.3, 5.8, 6.4.

Of course, you can index matrices and dataframes with longer vectors to get more data. Now, let's look at the first 8 rows of columns 1 and 3:

```
# Give me rows 1-8 and columns 1 and 3 of ToothGrowth
ToothGrowth[1:8, c(1,3)]

##    len dose
## 1  4.2  0.5
## 2 11.5  0.5
## 3  7.3  0.5
## 4  5.8  0.5
## 5  6.4  0.5
## 6 10.0  0.5
## 7 11.2  0.5
```

```
## 8 11.2  0.5
```

Figure 25: 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.

Get an entire row (or column)

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 of ToothGrowth
ToothGrowth[1, ]

##   len supp dose len.in index
## 1 4.2   VC  0.5 10.668     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 of ToothGrowth
ToothGrowth[, 2]
```

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.

Indexing dataframes with brackets [,]

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.VC.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.

subset()

subset()

subset()

`x`

The data (usually a dataframe)

`subset`

A logical vector indicating which rows you want to select

`select`

An optional vector of the columns you want to select

Figure 26: The `subset()` function is like a lightsaber. An elegant function from a more civilized age...

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`):

```
# Create a subsetted dataframe from ToothGrowth
#  of rows where len < 20, supp == "OJ" and dose >= 1

subset(x = ToothGrowth,
       subset = len < 20 &
                supp == "OJ" &
```

```
               dose >= 1
      )

##     len supp dose len.in index
## 41 19.7   OJ    1 50.038    41
## 49 14.5   OJ    1 36.830    49
```

As you can see, there were only two datapoints that satisfied all 3 of our subsetting 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 (see example on the right):

```
# EXPLANATION:
#
#  From the ToothGrowth dataframe,
#  give me all rows where len < 20
#  AND supp == "OJ" AND dose >= 1.
#  Then, return just the len and
#  supp columns.

subset(x = ToothGrowth,
       subset = len < 20 &
                supp == "OJ" &
                dose >= 1,
       select = c(len, supp)
       )

##     len supp
## 41 19.7   OJ
## 49 14.5   OJ
```

Combining slicing and dicing 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:

```
# 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] 20.66333
```

Or, we can do the same thing with 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] 20.66333
```

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

```
# Step 1: Take the mean of len column, indexed by
#  the supp column

mean(ToothGrowth$len[ToothGrowth$supp == "OJ"])

## [1] 20.66333
```

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

Additional tips

with()

`with(df, )`

The function `with()` 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, using the `ToothGrowth` dataset, we can calculate a vector defined as `len` divided by `dose` using the `with()` function where we specify the name of the dataframe (`ToothGrowth`) just once at the beginning of the line, and then refer to the column names without re-writing the dataframe or using the $ operator:

```
# Create a vector of len / dose from ToothGrowth
with(ToothGrowth, len / dose)

# THIS IS IDENTICAL TO

ToothGrowth$len / ToothGrowth$dose
```

If you find yourself making several calculations on one dataframe, the `with()` function can save you a lot of typing.

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	m	25	Superman	15
Sarina	m	25	Batman	12
Remon	m	29	Spiderman	12
Letizia	f	72	Batman	17
Babice	m	22	Antman	12
Jonas	f	35	Batman	9
Wendy	f	7	Superman	13
Niveditha	m	32	Maggott	875
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 30 whose favorite superhero is Spiderman?

Figure 27: 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.

7: Importing, saving, and managing data

Remember way back in Chapter 2[20] 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.

[20] 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.

Figure 28: Your workspace – all the objects, functions, and delicious glue you've defined in your current session.

Helpful workspace functions

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

Code	**Result**
`getwd()`	Returns the current working directory
`setwd(file = )`	Changes the working directory to a specified file location
`list.files()`	Returns a vector of all files and folders in the working directory
`ls()`	Display all objects in the current workspace
`rm(a, b, ..)`	Removes the objects `a`, `b`... from your workspace
`rm(list = ls())`	Deletes *all* objects in your workspace
`save(a, b, .., file = "myimage.RData)`	Saves objects `a`, `b`, ... to "myimage.RData"
`save.image(file = "myimage.RData")`	Saves your entire workspace to "myimage.RData" in the working directory
`load(file = "myimage.RData")`	Loads a stored workspace called "myimage.RData" from the working directory
`write.table(x, file = "mydata.txt")`	Saves the object x as a text file called "mydata.txt" to the working directory
`read.table(file = "mydata.txt")`	Reads a text file called "mydata.txt" in the working directory into R.

Workspace / Working Environment

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.

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. 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 in, or near, your working directory unless you say otherwise.

It is a good idea to have a new working directory for every R related project you work on. To see your current working directory, use the `getwd()`:

getwd()

```
# Print my current working directory
getwd()
```

```
## [1] "/Users/CaptainJack/Desktop/yarrr"
```

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`.

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:

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

setwd()

Project in RSTudio

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

will automatically be set to the directory that the `.RProj` file is located in. Personally, I recommend creating a new R Project whenever you are starting a new research project. However, it's not necessary.

Once you've decided on a working directory for a specific R project, it's a good idea to put new folders in the directory which will contain your R code, data files, notes, and other material relevant to your project. In Figure 29 you can see how my working directory looks for a project I am working on called ForensicFFT.

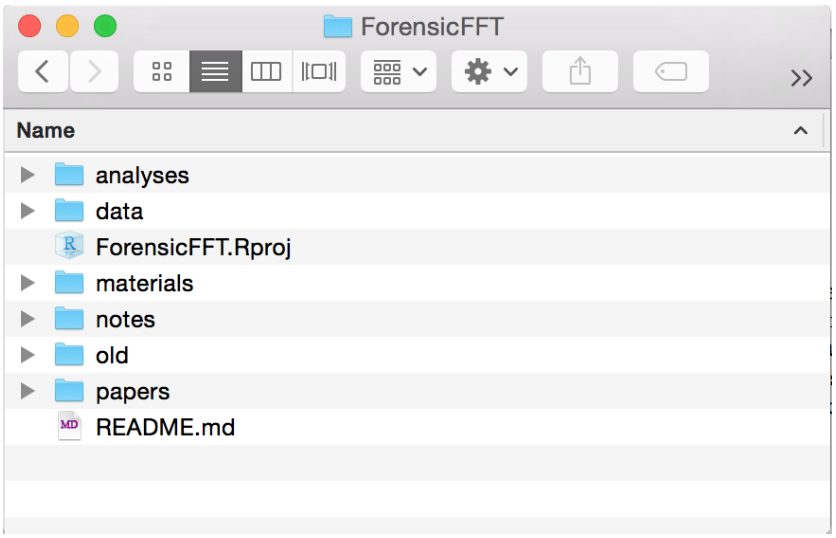


Figure 29: Here is the folder structure I use for the working directory in my R project called ForensicFFT. As you can see, it contains an .Rproj file generated by RStudio which sets this folder as the working directory. I also created separate folders for analyses (e.g.; R code), data (e.g.; text files, and .RData image files) and notes among others

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()
```

ls()

```
## [1] "experiment.df" "htest"         "a"             "b"
```

The result above says that I have 4 objects in my workspace called experiment.df, htest, a and b.

.RData files

When you save objects from R to your computer using the `save.image()` and `save()` functions, the objects are saved as `.RData` files. If you open a `.RData` file outside of R (say by double-clicking it), this will

open either R or RStudio (you can choose which by right-clicking) and load the objects it contains into a new workspace.

The `.RData` file format is specific to R and cannot be opened in any other software. If you want to save data like a dataframe in a format that can be read by other software, like Excel or SPSS, you'll need to save the object as a text file using a function like `write.table()`. We'll get to this a bit later.

save.image()

save.image()

To save all the objects in your workspace as a .RData file, use the `save.image()` function. This will save all the objects as a .RData file close to your working directory. For example, to save my workspace in the `data` folder located in my working directory, I'd run the following:

```
# Save my workspace to myimage.RData in th,e
#  data folder of my working directory

save.image(file = "data/myimage.RData")
```

save()

save()

If you don't want to save your entire workspace, but instead only want to save a few selected objects, you can use the `save()` function. For example, let's say I want to store both data from an experiment, and the results of a hypothesis test, in one file called `Mar7_experiment.RData`. If the data is stored in a dataframe called `experiment.df`, and the hypothesis test is stored in an object called `htest`, I can run the following code:

```
save(experiment.df, htest,
     file = "data/Mar7_experiment.RData")
```

load()

load()

To load a saved workspace, use the `load()` function. For example, to load the workspace that I just saved to the data folder in my working directory, I'd run the following:

```
load(file = "data/myimage.RData")
```

rm()

rm()

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 objects from your workspace, use the `rm()` function.

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())
```

Exporting dataframes as .txt files

write.table()

While `.RData` files are great for saving R objects, sometimes you'll want to export data (specifically dataframes) as a simple .txt text file that other programs, like Excel and SPSS, can also read. To do this, use the `write.table()` function.

write.table()

`x`

The object you want to write to a text file. Usually a dataframe object.

`file`

The document's file path relative to the working directory unless specified otherwise. For example `file = "mydata.txt"` will save the data as a text file directly in the working directory, while `file = "data/mydata.txt"` will save the data in a folder called 'data' inside the working directory. If you want to write the file to someplace outside of your workspace, just put the full file path.

`sep`

A string indicating how the columns are separated. For comma separated files, use `''','''`, for tab–delimited files, use `''\t''`

`row.names`

A logical value (T or F) indicating whether or not save the rownames in the text file.

For example, the following code will save the `ToothGrowth` object as a tab–delimited text file in my working directory:

```
# Write the ChickWeight dataframe object to a tab-delimited
#  text file called chickweight.txt in my working directory

write.table(x = ChickWeight,
            file = "chickweight.txt",
            sep = "\t"
            )
```

If you want to save a file to a folder inside of your working directory, just include the folder name before the file name. For example, if I have a folder called `data` in my working directory, I can save the file to this folder with the `file = "data/chickweight.txt"` argument. Finally, if you want to save a file to a location outside of your working directory, just use the entire directory name. For example, to save a text file to my Desktop, I would set `file = "Users/nphillips/Desktop"`. When you enter a long path name into the `file` argument of `read.table()`, R will look for that directory outside of your working directory.

Reading dataframes from .txt files

read.table()

If you have a text file that you want to read into R, use the `read.table()` function.

read.table()

`file`

The document's file path (make sure to enter as a string with quotation marks!) OR an html link to a file.

`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 `","`, for tab-delimited files, use `"\t"`

`stringsAsFactors`

A logical value indicating whether or not to convert strings to factors. I always set this to FALSE (because I don't like using factors)

The three critical arguments to `read.table()` are `file`, `sep`, and `header`. The `file` argument is a character value telling R where to find the file. This can be either a specific directory on your harddrive, or a folder in your working directory. 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"`. Finally, the `header` is a logical value (T or F) telling R whether or not the first row in the data is the name of the data columns.

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 into
#  R and store as a new object called mydata

mydata <- read.table(file = 'data/mydata.txt',
```

```
                    sep = '\t',
                    header = T
                    )
```

Again, if you want to read a test file outside of your working directory, just use the entire path as the argument to `file = `. For example, to read in a datafile on my desktop, I'd use the argument `file = "/Users/nphillips/Desktop/mydata.txt"`.

Reading datafiles directly from a web URL

A really neat feature of the `read.table()` function is that you can use it to load datafiles directly from the web. To do this, just set the file path to the document's web URL (beginning with `http://www.`). 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:

```
fromweb <- read.table(file = 'http://goo.gl/jTNf6P',
                      sep = '\t',
                      header = T
                      )

## Warning in file(file, "rt"): unable to resolve 'goo.gl'
## Error in file(file, "rt"): cannot open the connection
```

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.

The `fromweb` dataframe should look like this:

```
fromweb

## Error in eval(expr, envir,
## enclos): object 'fromweb' not found
```

Additional Tips

- 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.
- If you absolutely have to read a non-text file into R, check out the package called `foreign`. This package has functions for importing Stata, SAS and Shitty Piece of Shitty Shit files directly into R. To read Excel files, try the package `xlsx`

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[21]
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.

[21] 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.

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.

8: Advanced dataframe manipulation

In this chapter we'll cover some more advanced functions and procedures for manipulating dataframes

Row and column statistics: `rowMeans`, `colMeans`, `rowSums`, `colSums`

To calculate sums or means of numerical data across rows or columns of a matrix or dataframe, you can use one of the functions: rowMeans, `colMeans`, `rowSums` and `colSums`.

For example, let's say you give an exam containing 3 questions to 5 different students. The data could be represented in a matrix, where each row is a student and each column is a question as follows:

```
exam <- data.frame(
                   "q1" = c(1, 5, 2, 3, 2),
                   "q2" = c(8, 10, 9, 8, 7),
                   "q3" = c(3, 7, 4, 6, 4)
                   )

exam
```

```
##   q1 q2 q3
## 1  1  8  3
## 2  5 10  7
## 3  2  9  4
## 4  3  8  6
## 5  2  7  4
```

To calculate the mean score across *columns* (questions), we can use the `colMeans` function. This will return the mean score for each column.

```
colMeans(exam)
```

```
##  q1  q2  q3
## 2.6 8.4 4.8
```

It looks like question 1 was the hardest (with a mean score of 2.6), and question 2 was the easiest (with a mean score of 8.4).

To calculate the mean score across *rows* (students), we can use the `rowMeans` function. This will return the mean score for each row:

```
rowMeans(exam)

## [1] 4.000000 7.333333 5.000000 5.666667 4.333333
```

It looks like the second student did the best (with a mean score of 8.4).

As you can imagine, the `rowSums` and `colSums` functions work exactly the same way – except they calculate sums instead of means:

```
colSums(exam)

## q1 q2 q3
## 13 42 24

rowSums(exam)

## [1] 12 22 15 17 13
```

Sorting dataframes with `order()`

To sort the rows of a dataframe according to column values, use a combination of indexing, reassignment, and 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 it's order.

Let's re-order our exam data by the values of the question 1 column. To do this, we'll put `order(exam$q1)` as the row index:

Important! Don't forget than in order to change an object in R, you *must* reassign it! If you try to reorder a dataframe without reassigning it with the <- operator, it won't change.

```
# Sort the exam dataframe by Q1
exam <- exam[order(exam$q1),]
exam

##   q1 q2 q3
## 1  1  8  3
## 3  2  9  4
## 5  2  7  4
## 4  3  8  6
## 2  5 10  7
```

To order a dataframe by several columns, just add additional arguments to `order()`. For example, to order the dataframe by Question 1 *and then* by Question 2, we'd do the following:

```
# Sort the exam dataframe by Q1 and then Q2
exam <- exam[order(exam$q1, exam$q2),]
```

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 = T` to the `order()` function:

```
# Sort the exam dataframe by Q3 in decreasing order
exam <- exam[order(exam$q3, decreasing = T),]
```

You can also use the `order()` function to order character vectors in alphabetical order. For example, if I add a character vector of student names to the `exam` dataframe, I can then sort the data according to student in alphabetical order:

```
# Add student names to the exam dataframe
exam$student <- c("david", "jack", "vincent", "heath", "alice")

# Sort the exam data by the student names
exam <- exam[order(exam$student),]
exam

##   q1 q2 q3 student
## 1  1  8  3   alice
## 2  5 10  7   david
## 3  2  9  4   heath
## 4  3  8  6    jack
## 5  2  7  4 vincent
```

Grouped aggregation with aggregate()

In Chapter 6, you learned how to calculate statistics on subsets of data using subsetting, indexing, and basic functions. However, you may have noticed that the code is not terribly efficient. For example, let's say you want to know if a person's favorite TV show predicts whether or not they smoke. To answer this, you might conduct a study where you ask 1,000 people to name their favorite show and indicate whether or not they smoke. After you collect the data, you want to calculate, what percent of people smoke for every favorite TV show. If you try to do this with Chapter 6 techniques, you'd have to write a separate line of code for *every* possible TV show in the data. Thankfully, R has some great built-in functions that allow you to easily apply functions (like `mean()`) to a dependent variable (like whether or not people smoke) for *every* level of one or more independent variables (like their favorite TV show) with just a few lines of code.

```
# The SLOW way to calculate summary statistics

mean.seinfeld <- mean(smoke[tv == "seinfeld"])
mean.simpsons <- mean(smoke[tv == "simpsons"])
mean.tatort <- mean(smoke[tv == "tatort"])
#.....
```

The first aggregation function we'll cover is `aggregate()`. The function `aggregate()` takes three arguments, a formula in the form `y ~ x1 + x2 + ...` defining the dependent (Y) and one or more independent variables (x1, x2, ...), a function (FUN), and a dataframe (data). When you execute `aggregate(y ~ x1 + x2 + ..., data, FUN)`, R

will apply the input function (FUN) to the dependent variable (Y) *separately* for each level(s) of the independent variable(s) (x1, x2, ...) and then print the result. Let's see how it works:

aggregate()

`formula`

A formula in the form `y ~ x1 + x2 + ...` where y is the dependent variable, and x1, x2... are the index (independent) variables. For example, `salary ~ sex + age` will aggregate a `salary` column at every unique combination of age and age

`FUN`

A function that you want to apply to x at every level of the independent variables. For example, `FUN = mean` will calculate the mean for each level of the independent variables.

`data`

The dataset containing the variables in `formula`

`...`

Optional arguments passed on to FUN (like `na.rm = T` to ignore `NA values in x`)

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 `ToothGrowth` dataset to answer the question "What is the mean tooth length for each supplement?" For this question, we'll set the value of the dependent variable Y to `len`, x1 to `supp`, and FUN to `mean`

aggregate()

```
# Calculate the mean tooth length (DV) for
#  EVERY value of supp (IV)

aggregate(formula = len ~ supp,  # DV is len, IV is supp
          FUN = mean, # Calculate the mean of each group
          data = ToothGrowth # dataframe is ToothGrowth
          )

##   supp      len
## 1   OJ 20.66333
## 2   VC 16.96333
```

As you can see, the `aggregate()` function has returned a dataframe

with a column for the independent variable (supp), 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 get by manually indexing each level of supp individually.

Combine aggregate() with subset()

If you want to aggregate statistics for a subset of a dataframe, just subset the dataframe in the data argument with the subset() function:

```
# Calculate the median tooth length
#  for EVERY value of SUPP
#  ONLY for dose > .5

aggregate(formula = len ~ supp,
          FUN = median,
          data = subset(x = ToothGrowth,
                        subset = dose > .5)
          )
```

You can also include multiple independent variables in the formula argument to aggregate(). For example, let's use aggregate() to now get the median value of len for all combinations of both supp and dose (the amount of the supplement):

```
# Calculate the median tooth length
#  for every combination of supp AND dose

aggregate(formula = len ~ supp + dose,
          FUN = median,
          data = ToothGrowth
          )

##   supp dose   len
## 1   OJ  0.5 12.25
## 2   VC  0.5  7.15
## 3   OJ  1.0 23.45
## 4   VC  1.0 16.50
## 5   OJ  2.0 25.95
## 6   VC  2.0 25.95
```

Our new result now shows us the median length for all combinations of both supplement (supp) and dose.

You can even use aggregate() to evaluate functions that return more than one value. For example, the summary() function returns

several summary statistics from a vector. Let's use the `summary()` function in `aggregate()` to calculate several summary statistics for each combination of `supp` and `dose`:

```
# Calculate summary statistics of tooth length
#  for every combination of supp AND dose

aggregate(formula = len ~ supp + dose,
          FUN = summary,
          data = ToothGrowth
          )
```

```
##   supp dose len.Min. len.1st Qu. len.Median len.Mean len.3rd Qu. len.Max.
## 1   OJ  0.5     8.20        9.70      12.25    13.23       16.18    21.50
## 2   VC  0.5     4.20        5.95       7.15     7.98       10.90    11.50
## 3   OJ  1.0    14.50       20.30      23.45    22.70       25.65    27.30
## 4   VC  1.0    13.60       15.27      16.50    16.77       17.30    22.50
## 5   OJ  2.0    22.40       24.58      25.95    26.06       27.08    30.90
## 6   VC  2.0    18.50       23.38      25.95    26.14       28.80    33.90
```

While `aggregate()` is good for calculating summary statistics for a single dependent variable, it can't handle multiple dependent variables. For example, if you wanted to calculate summary statistics for multiple dependent variables in a dataset, you'd need to execute an `aggregate()` command for each dependent variable, and then combine the results into a single dataframe. Thankfully, a recently released R package called `dplyr` makes this process very simple!

Aggregation with the dplyr package

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 load it[22]

```
library(dplyr)
```

[22] If the dplyr package isn't installed on your computer, you'll need to install it by running `install.packages("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...'

The pipe operator %>% in dplyr is used to link multiple arguments sequentially. You can think of %>% 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 called `grouping.column`, and the columns you want to aggregate are called `col.a`, `col.b` and `col.c`

```
my.df %>%  # Specify original dataframe
    group_by(grouping.column) %>% # Grouping variable
    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 `ToothGrowth` dataset. I'm going to group the data according to the columns supp and dose. I'll then create several columns is a 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 `ToothGrowth.agg`:

```
ToothGrowth.agg <- ToothGrowth %>% # Define dataframe, THEN...
                group_by(supp, dose) %>% # Define the grouping variable, THEN...
  summarise( # Tell R you are going to calculate summaries
    len.min = min(len), # Define first summary...
    len.mean = mean(len),  # Define second summary...
    len.sd = sd(len),
   n = n() # How many are in each group?
  ) # End

ToothGrowth.agg

## Source: local data frame [6 x 6]
## Groups: supp [?]
##
##     supp  dose len.min len.mean   len.sd     n
##   (fctr) (dbl)   (dbl)    (dbl)    (dbl) (int)
```

```
## 1     OJ   0.5     8.2    13.23 4.459709    10
## 2     OJ   1.0    14.5    22.70 3.910953    10
## 3     OJ   2.0    22.4    26.06 2.655058    10
## 4     VC   0.5     4.2     7.98 2.746634    10
## 5     VC   1.0    13.6    16.77 2.515309    10
## 6     VC   2.0    18.5    26.14 4.797731    10
```

As you can see, our final object `ToothGrowth.agg` is the aggregated dataframe we want which aggregates all the columns we wanted for each combination of supp and dose. 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 create a new dataframe called `movies.agg` that aggregates data from the movies dataframe. In addition to aggregating the data, we'll filter out data to only include values that specify some criteria.

```
library(yarrr) # load the yarrr package

movies.agg <- movies %>% # From the movies dataframe...
            filter(genre != "Horror" & time > 50) %>% # Select only these rows
            group_by(rating, sequel) %>% # Group by genre 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?
            )

movies.agg # Print the result!

## Source: local data frame [14 x 6]
## Groups: rating [?]
##
##       rating sequel frequency budget.mean revenue.mean   billion.p
##        (chr)  (int)     (int)       (dbl)        (dbl)       (dbl)
## 1          G      0        59   56.564837    233.54391 0.000000000
## 2          G      1        12  111.500000    357.22634 0.083333333
## 3      NC-17      0         2    3.750000     18.48954 0.000000000
## 4  Not Rated      0        84   13.281818     55.52732 0.000000000
## 5  Not Rated      1        12    8.000000     66.06133 0.000000000
## 6         PG      0       312   64.885651    190.58432 0.009615385
## 7         PG      1        62   87.036364    371.67376 0.016129032
## 8      PG-13      0       645   59.893058    167.70012 0.006201550
## 9      PG-13      1       120  127.347863    523.79108 0.116666667
## 10         R      0       623   37.673610    109.14962 0.000000000
## 11         R      1        42   66.121622    226.16161 0.000000000
## 12        NA      0        86    8.332353     33.67170 0.000000000
## 13        NA      1        15   41.300000     48.07824 0.000000000
## 14        NA     NA        11         NaN     34.05020 0.000000000
```

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.

Additional Tips

- There is an entire class of functions in R that apply functions to groups of data. One common one is `tapply()`, `sapply()` and `lapply()` which 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,
                    INDEX = genre,
                    FUN = mean,
                    na.rm = T))
```

- We've only scratched the surface of what you can do with `dplyr`. For more tips, check out the dplyr vignette at https://cran.r-project.org/web/packages/dplyr/vignettes/databases.html.

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://dl.dropboxusercontent.com/u/7618380/datasets/caffeinestudy.txt

1. Load the dplyr library
2. Load the dataset from https://dl.dropboxusercontent.com/u/7618380/datasets/caffeinestudy.txt as a new dataframe called `caffeine`.
3. Calculate the mean age for each gender
4. Calculate the mean age for each drink
5. Calculate the mean age for each combined level of both gender and drink
6. Calculate the median score for each age
7. For men only, calculate the maximum score for each age
8. Create a dataframe showing, for each level of drink, the mean, median, maximum, and standard deviation of scores.
9. 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.

9: Sampling and Probability Distributio

Sampling data from probability distributions

By now you know how to generate sequences of numbers with the functions :, `seq()`, and `rep()`. However, these functions don't generate very interesting data. Instead, we can use R to generate randomly sampled data from specified *probability distributions*. A probability distribution is simply an equation – also called a likelihood function – that indicates how likely certain numerical values are to be drawn.

For example, imagine you need to hire a new group of pirates for your crew. You have the option of hiring people form 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 30 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 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.

In the next section we'll go over some of the most commonly used sampling distributions: the Normal and Uniform distributions.

```r
# Create blank plot
plot(1, xlim = c(0, 100), ylim = c(0, 100),
     xlab = "Pirate Quality", ylab = "", type = "n",
     main = "Two different Pirate colleges", yaxt = "n"
     )

# Set colors
require("RColorBrewer")
col.vec <- brewer.pal(10, name = "Set3")[4:6]

# Draw Samples
samples.1 <- runif(n = 5, 40, 60)
samples.2 <- runif(n = 5, 10, 90)

text(50, 90, "Pirate Training Unlimited", font = 3)

for(i in 1:length(samples.1)) {

  points(samples.1[i], 75, pch = 21, bg = col.vec[1], cex = 3)
  text(samples.1[i], 75, round(samples.1[i], 0))

}

segments(40, 65, 60, 65, col = col.vec[1], lty = 1, lwd = 2)

text(50, 40, "Unlimited Pirate Training", font = 3)

for(i in 1:length(samples.2)) {

  points(samples.2[i], 25, pch = 21, bg = col.vec[2], cex = 3)
  text(samples.2[i], 25, round(samples.2[i], 0))

}

segments(10, 15, 90, 15, col = col.vec[2], lty = 1, lwd = 2)
```

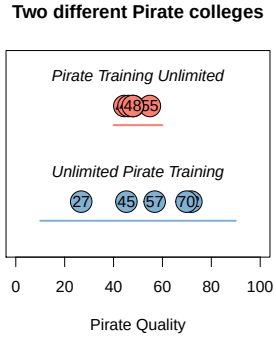


Figure 30: Sampling 5 potential pirates from two different pirate colleges. Pirate Training Unlimited (PTU) consistently produces average pirates (with scores between 40 and 60), while Unlimited Pirate Training (UPT), produces a wide range of pirates from 0 to 100.

The Normal (Gaussian) distribution

Let's start with the most famous distribution in statistics: the Normal (or if you want to sound pretentious, the Gaussian) distribution. The Normal distribution is bell-shaped, and has two parameters: a mean and a standard deviation. See the margin figure 31 for plots of three different Normal distributions with different means and standard deviations.

To generate samples from a normal distribution in R, we use the function `rnorm()` this function has three arguments:

`rnorm()`

`n`

The number of observations

`mean`

The mean of the Normal distribution from which samples are drawn (not the sample mean!!)

`sd`

The standard deviation of the Normal distribution from which samples are drawn

For example, let's draw 5 samples (`n = 5`) from a normal distribution with mean 0 (`mean = 0`) and standard deviation 1 (`sd = 1`)

```
rnorm(n = 5, mean = 0, sd = 1)

## [1]  0.1756358  0.6535578 -0.0654903  0.6455222  0.3642940
```

This code returns a vector of 5 values, where each value is a new random sample drawn from a Normal distribution with mean = 0 and standard deviation = 1.

Because the sampling is done randomly, you'll get different values each time you run the `rnorm()` (or any other random sampling) function. To see this, let's create two different sets of samples from a normal distribution with mean 10 and standard deviation 5 and see how they compare:

```
rnorm(5, mean = 10, sd = 5)

## [1] 14.217208  6.617606  8.042835 12.024703  4.465707
```

```
require("RColorBrewer")

# Create blank plot
plot(1, xlim = c(-5, 5), ylim = c(0, 1),
     xlab = "x", ylab = "dnorm(x)", type = "n",
     main = "Three Normal Distributions"
     )

# Set up design matrix for loop
design.matrix <- data.frame("mean" = c(0, -2, 1),
                            "sd" = c(1, .5, 2)
                            )
# Set colors
col.vec <- brewer.pal(10, name = "Set3")[4:6]

# Start loop over distributions
for (i in 1:nrow(design.matrix)) {

mean.i <- design.matrix$mean[i]
sd.i <- design.matrix$sd[i]

fun <- function(x) {
  dnorm(x, mean = mean.i, sd = sd.i)}

curve(expr = fun,
      from = -5,to = 5,
      xlab = "x", lwd = 3,
      add = T, col = col.vec[i])

samples <- rnorm(n = 10, mean = mean.i, sd = sd.i)

segments(x0 = samples, y0 = rep(0, 10),
         x1 = samples, y1 = fun(samples),
         col = col.vec[i], lwd = 1, lty = 2
         )
}

legend.fun <- function(i) {
  paste("mean = ", design.matrix$mean[i],
        ", sd = ", design.matrix$sd[i], sep = "")}

legend("topright",
       legend = c(legend.fun(1),
                  legend.fun(2),
                  legend.fun(3)
                  ),
       lwd = rep(3, 3),
       col = col.vec[1:3]
       )
```

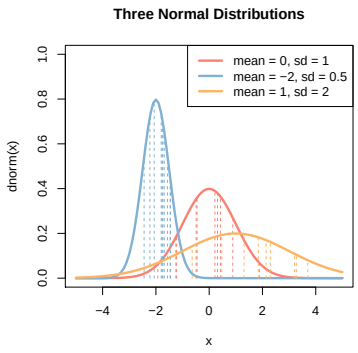


Figure 31: Three different normal distributions with different means and standard deviations.

```r
rnorm(5, mean = 10, sd = 5)
```

```
## [1]  8.083498 11.470101 18.965303 14.070161  3.399878
```

As you can see, even though I used the exact same code to generate the vectors a and b, the numbers in each sample are different. That's because the samples are each drawn randomly and independently from the normal distribution. To visualize the sampling process, run the code in the margin Figure 31 on your machine several times. You should see the sampling lines dance around the distribution.

```r
# uniform distribution
curve(dunif,
      from = 0, to = 1,
      xlim = c(-.5, 1.5),
      xlab = "x",
      lwd = 2,
      main = "Uniform\nmin = 0, max = 1")
```

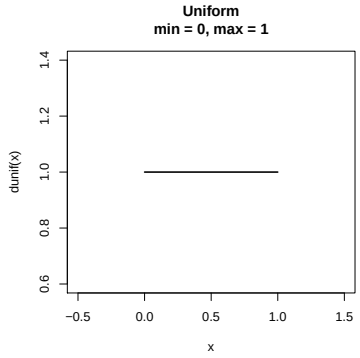


Figure 32: The Uniform distribution - known colloquially as the Anthony Davis distribution.

The Uniform distribution

Next, let's move on to the *uniform* distribution. The uniform distribution gives equal probability to all values between the minimum and maximum values.

To generate samples from a uniform distribution, we use the function `runif()`, the function has 3 arguments:

`runif()`

`n`

The number of observations (i.e.; samples)

`min`

The lower bound of the Uniform distribution from which samples are drawn

`max`

The upper bound of the Uniform distribution from which samples are drawn

Let's draw 5 samples from two uniform distributions, one with bounds at 0 and 1, and one with bounds at -100 and 100:

```r
runif(5, min = 0, max = 1) # 5 samples from U[0, 1]
```

```
## [1] 0.03176634 0.57970549 0.15420484 0.12527050 0.14798581
```

```r
runif(5, min = -100, max = 100) # 5 samples from U[-100, 100]
```

```
## [1]  82.668526 -48.513919   8.493059  29.804493 -70.161126
```

Sampling from a set of values: sample()

The next function we'll use is **sample()**. The sample function works a bit differently from `runif()` and `rnorm()` because it allows to you to define which values you want to sample and the probability associated with each value. For example, if you want to simulate the flip of a fair coin, you can tell the sample function to draw the value "Heads" with probability .50, and the value "Tails" with probability .50.

sample()

`x`

A vector of outcomes you want to sample from. For example, to simulate coin flips, you'd enter `x = c("Heads", "Tails")`

`size`

The number of samples you want to draw.

`replace`

Should sampling be done with replacement? If T, then each individual sample will be replaced in the data vector. If F, then the same outcome will never be drawn more than once. Think about replacement like drawing different balls from a bag. Sampling with replacement (`replace = T`) means that each time you draw a ball, you return the ball back into the bag before drawing another ball. Sampling without replacement (`replace = F`) means that after you draw a ball, you remove that ball from the bag before drawing again.

`prob`

A vector of probabilities of the same length as `x` indicating how likely each outcome in "x" is. For example, to sample equally from two outcomes, you'd enter `prob = c(.5, .5)`. The first value corresponds to the first value of x and the second corresponds to the second value (etc.). The vector of probabilities you give as an argument should add up to one. However, if they don't, R will just rescale them so that they will sum to 1.

As a simple example, let's simulate 10 flips of a fair coin, were the probably of getting either a Head or Tail is .50:

```r
sample(x = c("Heads", "Tails"), # The values you want to sample from
       size = 10,  # The number of samples
       prob = c(.5, .5), # The probability of each value
       replace = T # Sampling with replacement
       )

##  [1] "Heads" "Tails" "Heads" "Heads" "Heads" "Tails" "Tails" "Tails"
##  [9] "Heads" "Tails"
```

As you can see, our function returned a vector of 10 values corresponding to our sample size of 10. Keep in mind that, just like using `rnorm()` and `runif()`, the `sample()` function can give you different outcomes every time you run it.

Drawing coins from a treasure chest

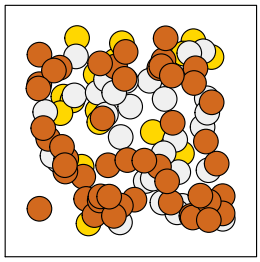
Now, let's sample drawing coins from a treasure chest Let's say the chest has 100 coins: 20 gold, 30 silver, and 50 bronze. Let's draw 10 random coins from this chest. Because we remove coins when we draw them, we'll set `replace = F`.

```r
# Create chest with the 100 coins

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

# Draw 10 coins from the chest without replacement

sample(x = chest,
       size = 10,
       prob = rep(1 / 100, times = 100),
       replace = F
       )

##  [1] "bronze" "silver" "bronze" "bronze" "bronze" "bronze" "silver"
##  [8] "silver" "silver" "bronze"
```

The output of the `sample()` function above is a vector of 10 strings indicating the type of coin we drew on each sample. The order of these strings matter: the first one is the first coin we drew, and the last one is the 10th coin we drew. 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...

```r
par(mar = c(3, 3, 3, 3))
plot(1, xlim = c(0, 1), ylim = c(0, 1),
     xlab = "", ylab = "", xaxt = "n",
     yaxt = "n", type = "n",
     main = "Chest of 20 Gold, 30 Silver,\nand 50 Bronze Coins")

points(runif(100, .1, .9),
       runif(100, .1, .9),
       pch = 21, cex = 3,
       bg = c(rep("gold", 20),
              rep("gray94", 30),
              rep("chocolate", 50))
       )
```

Chest of 20 Gold, 30 Silver, and 50 Bronze Coins



Simulating Pinder Outcomes

Let's simulate some Pinder outcomes. For those who don't know, Pinder is an app that allows Pirates to view profiles of potential dates. For each potential date, you can see their picture and either "like" them by swiping right, or "dislike" them by swiping left. If a pirate that you "liked" also "likes" you, then you've had a successful match and will be able to start chatting. let's say you "swipe right" on 20 Pinder profiles and the probability you get a match is 20%. We can simulate this using the sample function

```
sample(x = c("Match!!!", "No Match"),
       size = 20,
       replace = T, # Replace each sample back to the set
       prob = c(.2, .8)  # Probability of Match! is .2, and No Match :( is .8)
       )

##  [1] "No Match" "No Match" "No Match" "No Match" "No Match" "No Match"
##  [7] "No Match" "No Match" "No Match" "No Match" "No Match" "No Match"
## [13] "No Match" "No Match" "No Match" "Match!!!" "No Match" "No Match"
## [19] "No Match" "No Match"
```

The output of this function is a simulated response from 10 pirates that you liked.

Figure 33: A typical Pinder profile.

A worked example: A quick test of the law of large numbers

According to the law of large numbers, the larger our sample size, the closer our sample mean should be to the population mean. In other words, the more data (samples) you have, the more accurate your estimate should be. Let's test this by drawing either a small (N = 10) or a large (N = 1,000,000) number of observations from a Normal distribution with mean = 100 and sd = 20:

Tip: You can easily write large powers of 10 by using the notation `1eN`, where N is the power of 10. For example: `1e6` is the same as 1,000,000

```
small <- rnorm(10, mean = 100, sd = 20) # 10 observations
large <- rnorm(1e6, mean = 100, sd = 20) # One million observations
```

If our test worked, then the difference for the small sample should be larger than the large sample. Let's test this by calculating the mean of each sample and see how close they are to the true population mean of 100:

```
mean(small) # What is the mean of the small sample?

## [1] 100.9499

mean(large) # What is the mean of the large sample?

## [1] 100.0289

mean(small) - 100 # How far is the mean of Small from 100?
```

```
## [1] 0.9499404

mean(large) - 100 # How far is the mean of Large from 100?

## [1] 0.02889741
```

Looks like the law of large numbers holds up!

Test your R might!

1. Create a vector `samp1` with 100 values drawn from a Normal distribution with a mean of 100 and a standard deviation of 20. What is the mean and median of the samples? What percent of the samples are greater than 100?
2. Create a vector called `samp2` with 50 values drawn from a Uniform distribution with a minimum of -100 and a maximum of +100. What is the What is the mean and median of the samples? What percent of the samples are greater than 50?
3. Above all else, pirates love two things: treasure and gummy bears. There are few things a pirate likes more than reaching into a bag of fresh gummy bears and pulling out a handful of those gifts from the pirate gods. Imagine that you have a bag filled with 100 gummy bears: 30 Green, 40 Red, 10 Blue, and 20 Puce. Draw a random sample of 10 gummy bears from this bag. How many of each kind did you get in the sample? What percent of the sample are the terrible vomit-flavored puce color?

Figure 34: A gummy captain.

Additional Tips

- Sometimes you may want sampling functions like `rnorm()` and `runif` to return consistent values for demonstration purposes. For example, let's say you are showing someone how to use the `rnorm()` function, and you want them get the same values you get. You can accomplish this by using the `set.seed()` function. To use the function, enter an integer as the main argument - it doesn't matter what integer you pick. When you do, you will fix R's random number generator to a certain state associated with the specific integer you use. After you run `set.seed()`, the next random number generation function you use will always give the same result. Let's try generating 5 values from a Normal distribution with mean = 10 and sd = 1. But first, I'll set the seed to 10.

Use `set.seed()` to control what your random sampling functions return

```
set.seed(10)
rnorm(n = 5, mean = 10, sd = 1)

## [1] 10.018746  9.815747  8.628669  9.400832 10.294545
```

The result doesn't look so interesting, but watch me get the same result again by running the same code:

```
set.seed(10)
rnorm(n = 5, mean = 10, sd = 1)

## [1] 10.018746  9.815747  8.628669  9.400832 10.294545
```

Thankfully the change isn't permanent. If you run a sampling function twice after setting the seed once, the next run will be a new random sample.

10: Plotting: Part 1

Figure 35: 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.

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.

How does R manage plots?

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.

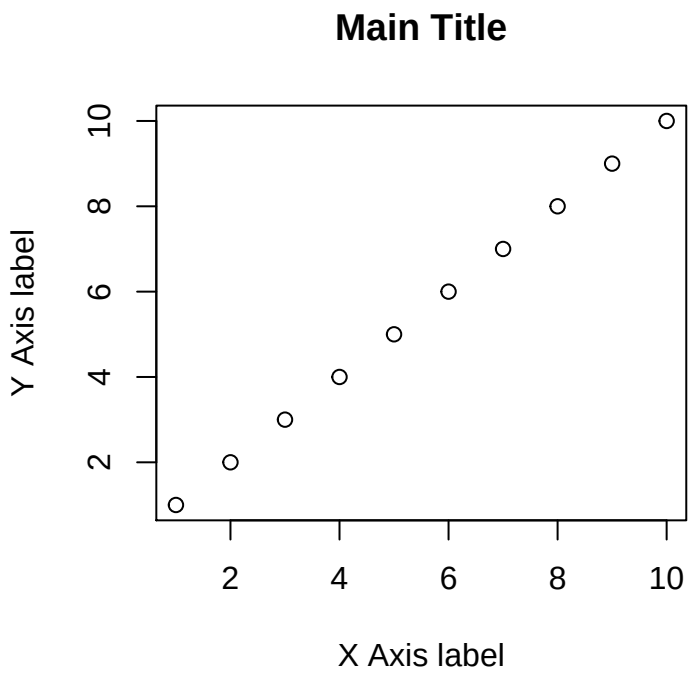
You can think of these plotting locations as canvases. You only have one canvas active at any given time, and any plotting command you run will put more plotting elements on your active canvas. Certain *high–level* plotting functions like `plot()` and `hist()` create brand new canvases, while other `low--level` 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"
    )
```



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.

Color basics

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.

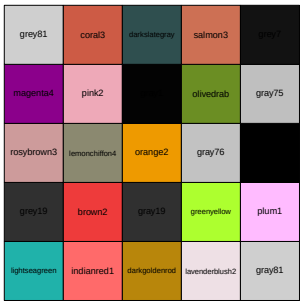
Specifying simple colors

`col = "red", col = "blue", col = "lawngreen" (etc.)`

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`. To see all color names in R, run the code

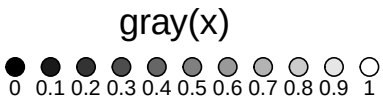
```
colors() # Show me all the color names!
```

Here are some random colors that are stored in R, try running `colors()` to see them all!



`col = gray(level, alpha)`

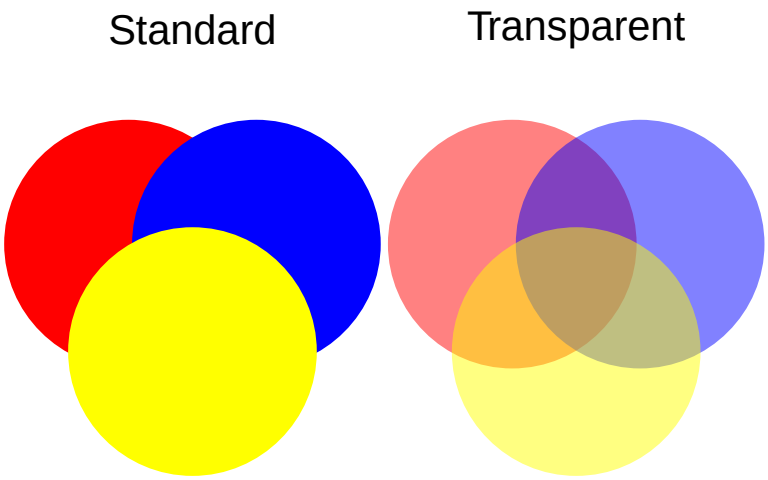
The `gray()` function takes two arguments, level and alpha, and returns a shade of gray. For example, `gray(level = 1)` will return white. The alpha argument specifies how transparent to make the color on a scale from 0 (completely transparent), to 1 (not transparent at all). The default value for alpha is 1 (not transparent at all.)



Transparent Colors with `transparent()`

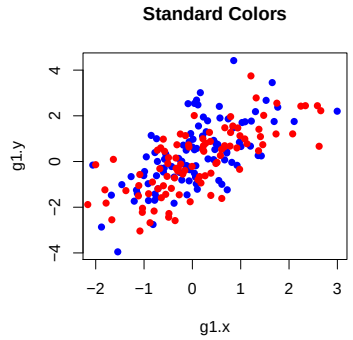
I don't know about you, but I almost always find transparent colors to be more appealing than solid colors. Not only do they help you see when multiple points are overlapping, but they're just much nicer to look at. Just look at the overlapping circles in the plot below.

```
par(mar = c(0, 0, 0, 0))
plot(1, xlim = c(0, 6), ylim = c(0, 1),
     type = "n", xaxt = "n", yaxt = "n",
     xlab = "", ylab = "", bty = "n")

text(1.5, .9, "Standard", cex = 1.5)

points(x = c(1, 2, 1.5),
       y = c(.5, .5, .3),
       col = c("red", "blue", "yellow"),
       pch = 16, cex = 20)

text(4.5, .9, "Transparent", cex = 1.5)

library(yarrr)

points(x = c(4, 5, 4.5),
       y = c(.5, .5, .3),
       col = c(transparent("red", .5),
               transparent("blue", .5),
               transparent("yellow", .5)),
       pch = 16, cex = 20)
```



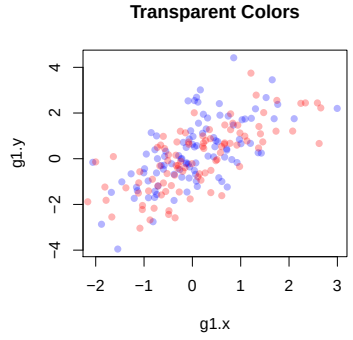

```
# Generate Random data
g1.x <- rnorm(100)
g1.y <- g1.x + rnorm(100)

g2.x <- rnorm(100)
g2.y <- g2.x + rnorm(100)

# Plot with Standard Colors
plot(g1.x, g1.y,
     col = "blue",
     pch = 16,
     main = "Standard Colors",
     cex = 1)

points(g2.x, g2.y,
       col = "red",
       pch = 16)
```

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`). You can either save the transparent color as a new color object, or use the function directly in a plotting function like I do in the scatterplot in the margin. Here are some examples:




```
library(yarrr)  # load the yarrr package

# Make my.blue, a transparent version of blue
my.blue <- transparent(orig.col = "blue",
                       trans.val = .5)

# Use my.blue as a color
plot(1, col = my.blue)

# Or put the transparent function in the plotting
#   function directly:

plot(1, col = transparent("blue", .5))
```

```
# Plot with Transparent  olors

library(yarrr) # Load the yarrr package

plot(g1.x, g1.y,
     col = transparent("blue", .7),
     pch = 16, main = "Transparent Colors",
     cex = 1)

points(g2.x, g2.y,
       col = transparent("red", .7),
       pch = 16)
```



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).

High vs. low-level plotting commands

There are two general types of plotting commands in R: high and low-level. High level plotting commands, like `plot()`, `hist()` and `pirateplot()` create entirely new plots. Within these high level plotting commands, you can define the general layout of the plot - like the axis limits and plot margins.

Low level plotting commands, like `points()`, `segments()`, and `text()` add elements to existing plots. These low level commands don't change the overall layout of a plot - they just add to what you've already created. Once you are done creating a plot, you can export the plot to a pdf or jpeg using the `pdf()` or `jpeg()` functions. Or, if you're creating documents in Markdown or Latex, you can add your plot directly to your document.

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.

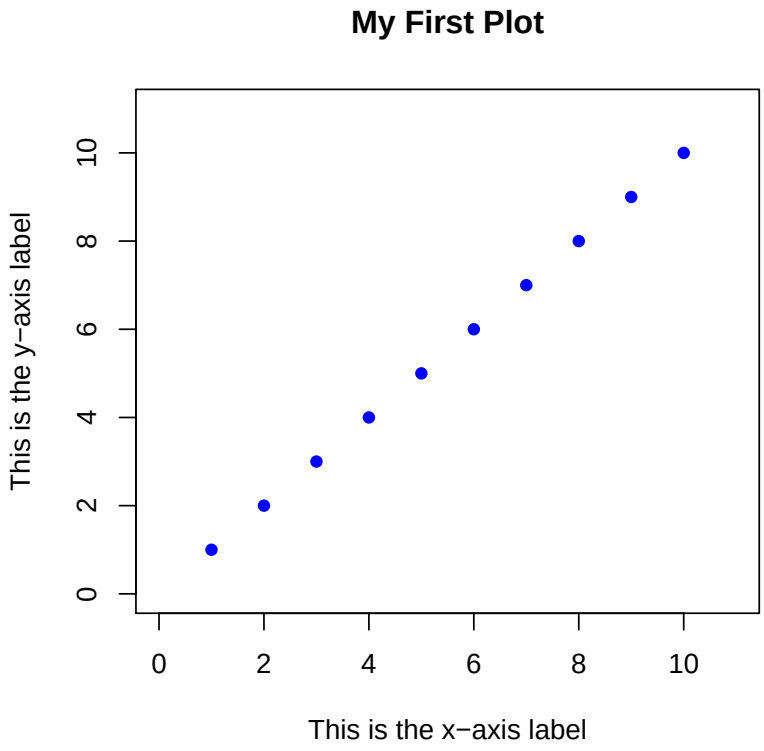
In the following examples, I will to cover the main plotting parameters for each plotting type. However, the best way to learn what you can, and can't, do with plots, is to try to create them yourself!

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.

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.

```
plot(x = 1:10, # x-coordinates
     y = 1:10, # y-coordinates
     type = "p", # Draw points (not 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

Here are some of the main arguments to `plot()`

`x, y` Vectors 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` Label for the plot title

`xlab` Label for the x-axis

`ylab` Labels for the y-axis

`xlim` Limits to the x-axis. For example, `xlim = c(0, 100)` will set the minimum and maximum of the x-axis to 0 and 100.

`ylim` Limits to the y-axis. For example, `ylim = c(50, 200)` will set the minimum and maximum of the y-axis to 50 and 200.

`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.

`bg` Color of the background of two-color symbols 21 through 25. For example `pch = 21, bg = "blue"` will the background of the two-color circle to Blue.

`cex` A numeric vector specifying the size of the symbols (from 0 to Inf). The default size is 1. `cex = 2` will make the points very large, while `cex = .5` will make them very small.

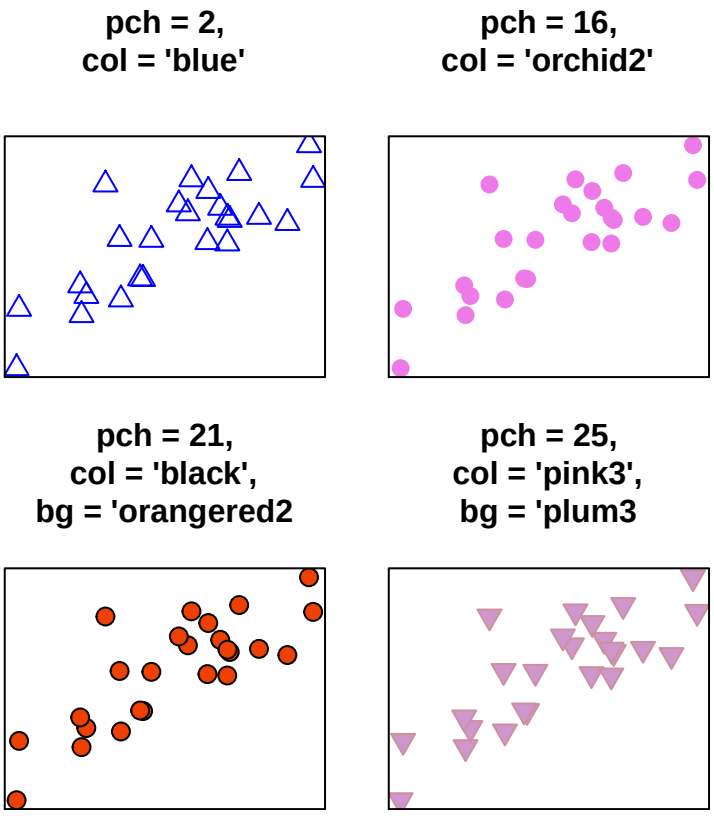
default value, or try to come up with a value that makes sense. For example, if you don't specify the `xlim` and `ylim` arguments, R will set the limits so that all the points fit inside the plot.

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 36 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:




```
par(mar  = c(1, 1, 3, 1))

plot(x = rep(1:5 + .1, each = 5),
     y = rep(5:1, times = 5),
     pch = 1:25,
     xlab = "", ylab = "", xaxt = "n", yaxt = "n",
     xlim = c(.5, 5.5),
     ylim = c(0, 6),
     bty = "n", bg = "gray", cex = 1.4,
     main = "pch = "
     )

text(x = rep(1:5, each = 5) - .35,
     y = rep(5:1, times = 5),
     labels = 1:25, cex = 1.2
     )
```

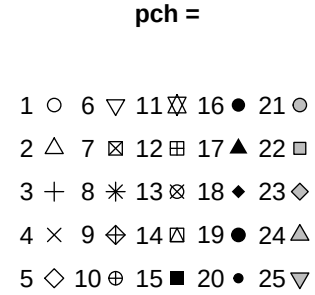


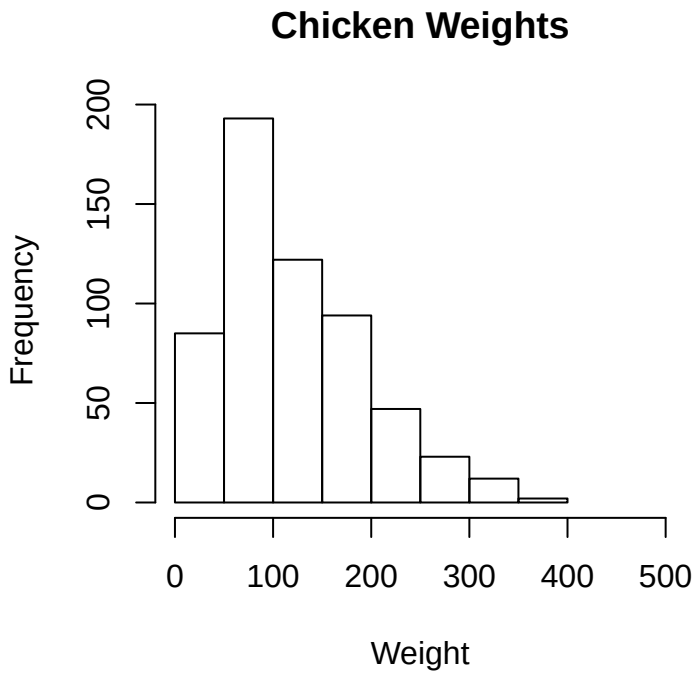
Figure 36: The symbol types associated with the `pch` plotting parameter.

Histogram: `hist()`

Histograms are the most common way to plot unidimensional 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 (see the margin figure 37)

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 = T` argument to the second plot. See Figure 38 to see this in action.

```
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
     )
```

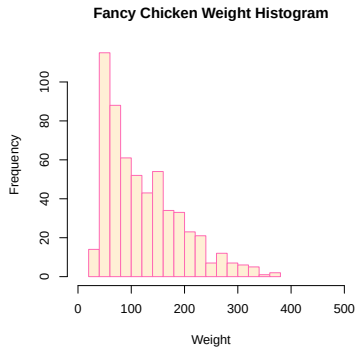


Figure 37: Fancy histogram

You can plot two histograms in one plot by adding the `add = T` argument to the second `hist()` function:

```
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 = T, # Add plot to previous one!
     col = gray(1, .5)
)
```

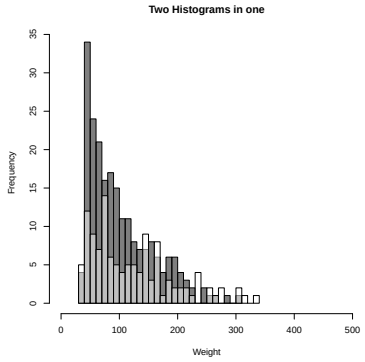
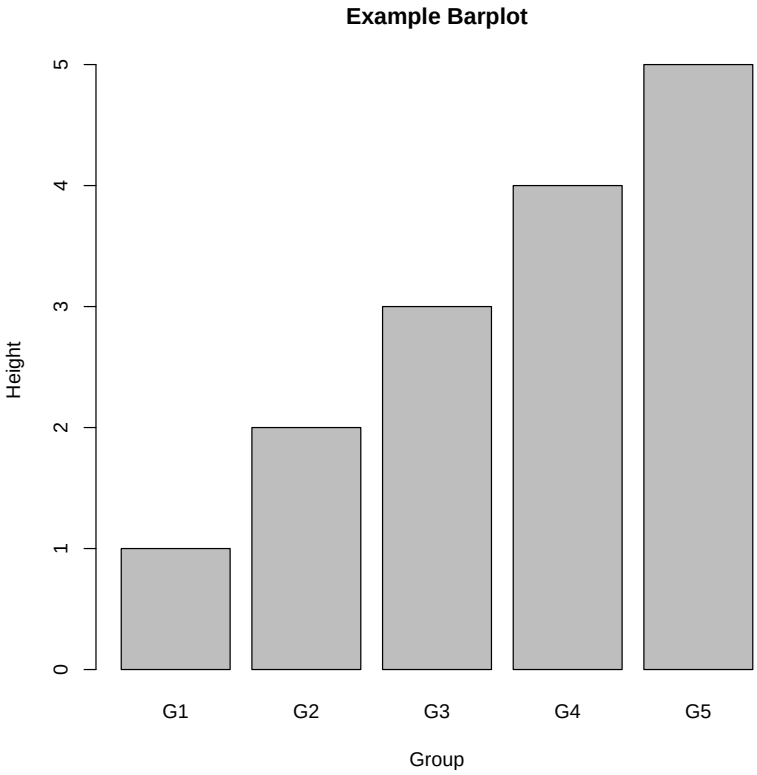


Figure 38: Plotting two histograms in the same plot using the `add = T` argument to the second plot.

Barplot: `barplot()`

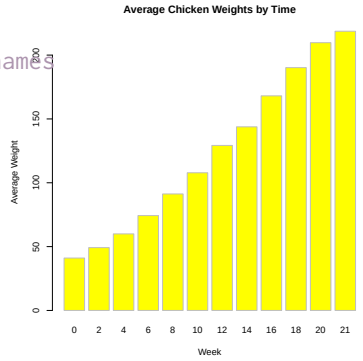
A barplot is good for showing summary statistics for different groups. The primary argument to a barplot is `height`: a vector of numeric values which will generate the height of each bar. To add names below the bars, use the `names.arg` argument. For additional arguments specific to `barplot()`, look 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 margin figure, I create a barplot with the average weight of chickens for each time point with yellow bars. If you want to plot different colored bars from different datasets, create one normal barplot, then create a second barplot with the add = T argument. In Figure 39, I plotted the average weights for chickens on Diets 1 and 2 separately on the same plot.

```
# Step 1: calculate the average weight by time
avg.weights <- aggregate(weight ~ Time,
                         data = ChickWeight,
                         FUN = mean)

# Step 2: Plot the result
barplot(height = avg.weights$weight,
        names.arg = avg.weights$Time,
        xlab = "Week",
        ylab = "Average Weight",
        main = "Average Chicken Weights by Time",
        col = "yellow",
        border = "gray")
```




```
# Calculate and plot average weights for Diet 2
d2.weights <- aggregate(weight ~ Time,
                        data = ChickWeight,
                        subset = Diet == 2,
                        FUN = mean)

barplot(height = d2.weights$weight,
      names.arg = d2.weights$Time,
      xlab = "Week",
      ylab = "Average Weight",
      main = "Average Chicken Weights by Time",
      col = transparent("green", .5),
      border = NA)

# Do the same for Diet 1 and add to existing plot
d1.weights <- aggregate(weight ~ Time,
                        data = ChickWeight,
                        subset = Diet == 1,
                        FUN = mean)

barplot(height = d1.weights$weight,
        add = T,  # Add to existing plot!
        col = transparent("blue", .5),
        border = NA)
```

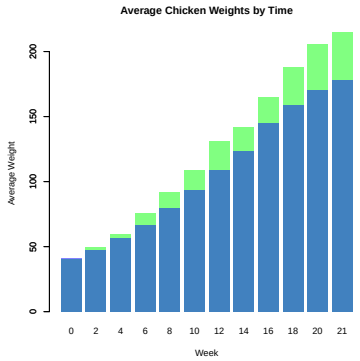


Figure 39: Stacked barplots by adding an additional barplot with the `add = T` argument.

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:

Table 1: Mean Swimming times (in meters / second)

	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()`

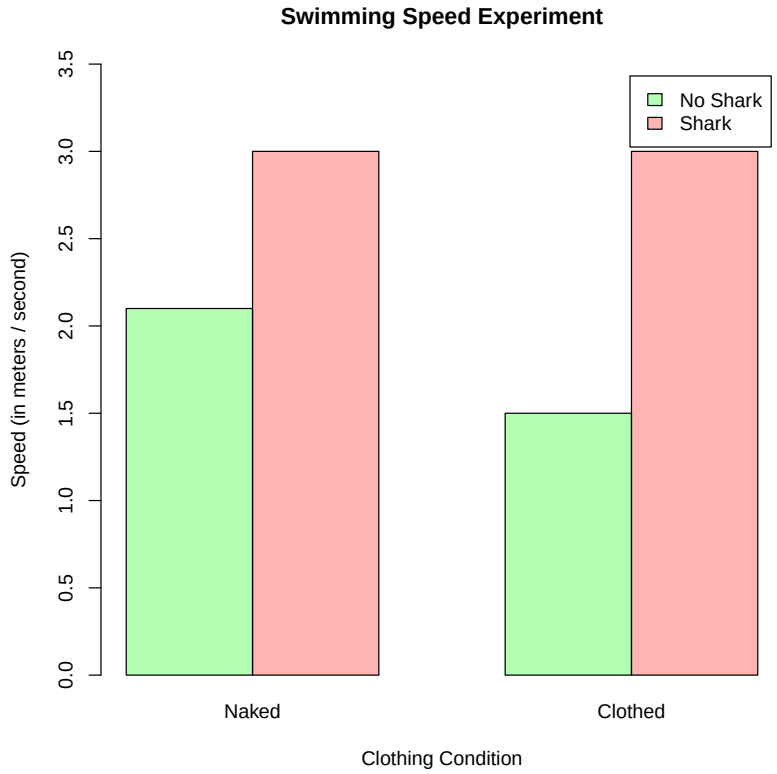
```
swim.data <- cbind(c(2.1, 3), # Naked Times
                   c(1.5, 3)) # Clothed Times

colnames(swim.data) <- c("Naked", "Clothed")
rownames(swim.data) <- c("No Shark", "Shark")
```

Here's how the final matrix looks:

```
swim.data

##          Naked Clothed
## No Shark   2.1     1.5
## Shark      3.0     3.0
```

Now, when I enter this matrix as the `height` argument to `barplot()`, I'll get multiple bars.

```
barplot(height = swim.data,
        beside = T,  # Put the bars next to each other
        legend.text = T,  # Add a legend
        col = c(transparent("green", .7),
                transparent("red", .7)),
        main = "Swimming Speed Experiment",
        ylab = "Speed (in meters / second)",
        xlab = "Clothing Condition",
        ylim = c(0, 3.5))
```


Swimming Speed Experiment
No Shark
Shark
Speed (in meters / second)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
Naked
Clothed
Clothing Condition

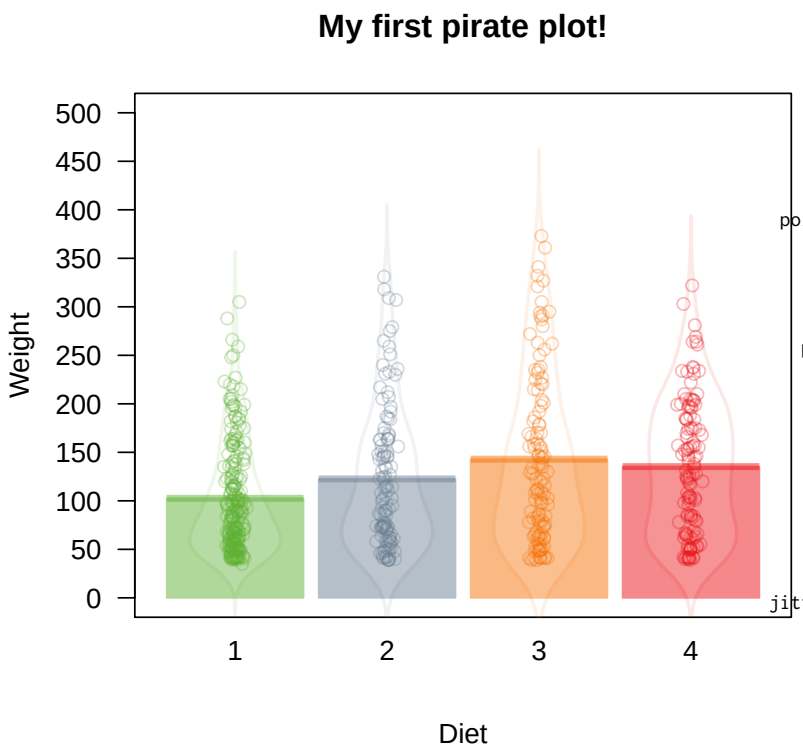
The Pirate Plot: `pirateplot()`

A Pirate Plot is the plotting choice of R pirates who want to plot a numeric dependent variable (like beard length) for each level of a categorical independent variable (like gender or favorite Spice Girl).

The two main arguments to `pirateplot()` are `formula` and `data`. In `formula`, you specify plotting variables in the form `dv ~ iv`, where `dv` is the name of the dependent variable, and `iv` is the name of the independent variable. In `data`, you specify the name of the dataframe object where the variables are stored.

I'll create a pirateplot of the ChickWeight data. I'll set the dependent variable to weight, and the independent variable to Diet.

```
library("yarrr")
pirateplot(formula = weight ~ Diet, # dv is weight, iv is Diet
           data = ChickWeight,
           main = "My first pirate plot!",
           xlab = "Diet",
           ylab = "Weight")
```



There are many additional optional arguments to `pirateplot()` – for example, you can use different color palettes with the `pal` argu-

The `pirateplot()` function is part of the yarrr package. So to use it, you'll first need to install the yarrr package

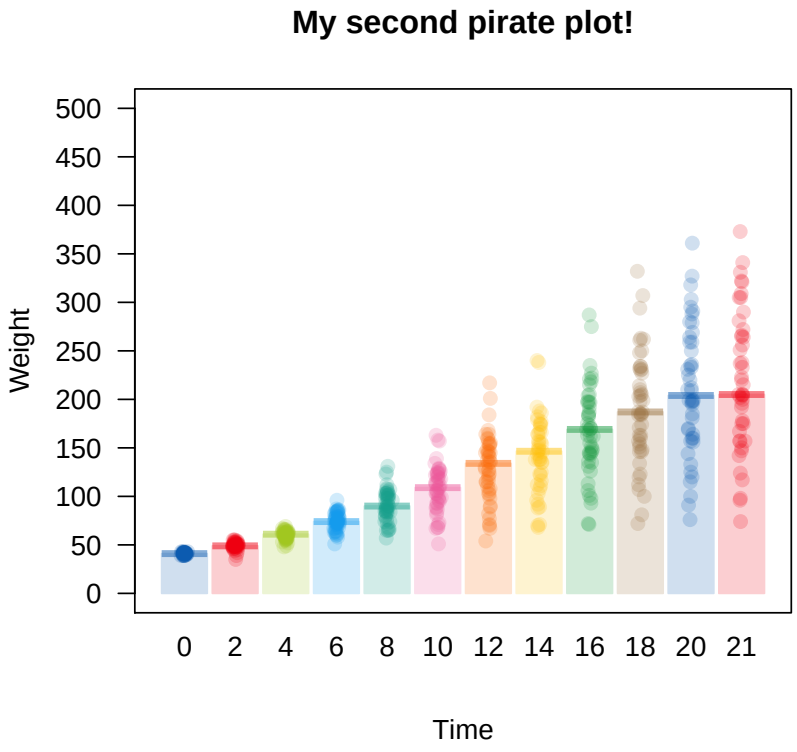
Here are some of the main arguments to `pirateplot()`

`formula`	A formula in the form `dv ~ iv1 + iv2` where where `dv` is the name of the dependent variable, and `iv1` and `iv2` are the name(s) of the independent variable(s). Up to two independent variables can be used.
`data`	A dataframe containing the dependent and independent variables.
`line.fun`	A function that determines the height of the lines and bars. Defaults to `mean`
`pal`	A string indicating the color palette of the plot. Can be a single color, or the name of a palette in the `piratepal()` function (e.g.; `"basel"`, `"google"`, `"drugs"`)
`point.cex`	A number indicating the size of the points
`bar.o`	A number between 0 and 1 specifying the opacity of the bars
`point.o`	... opacity of the points
`hdi.o`	... opacity of the 95% Highest Density Interval (HDI). Note that calculating HDIs can be time-consuming!
`line.o`	... opacity of the average line (i.e. the top of each bar).
`bean.o`	... opacity of the beans.
`jitter.val`	A number indicating how much to jitter the points (default is .05)
`...`	Additional arguments passed to the plot (e.g.; `main`, `xlab`, `ylab`, `xlim`, `ylim`)

ment or change how transparent different elements are with `bar.o`, `bean.o`, `line.o`, `point.o`, `hdi.o`. To see them all, look at the help menu with `?pirateplot`.

For example, here's a second pirateplot where I put Time on the x-axis, change the height of the bars to `median`, added a gray background

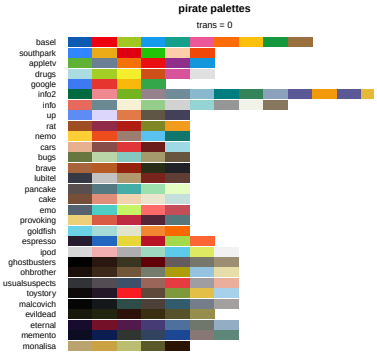
```
pirateplot(formula = weight ~ Time, # dv is weight, iv is Diet
           data = ChickWeight,
           line.fun = median, # median heights
           bean.o = 0, # No bean
           point.pch = 16, # Solid points
           point.o = .2,
           point.cex = 1.2, # Larger points
           pal = "basel",  # Basel palette
           back.col = gray(.97), # gray background
           bar.o = .2, # Lighter bar
           main = "My second pirate plot!",
           xlab = "Time",
           ylab = "Weight")
```



You can include up to two independent variables in the formula

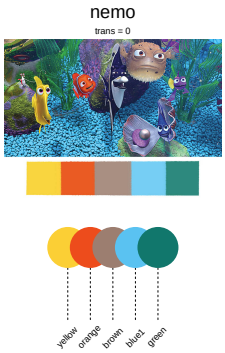
To see all the different palettes you can use in the `piratepal()` function, run the following code:

```
piratepal(palette = "all",
          action = "show")
```



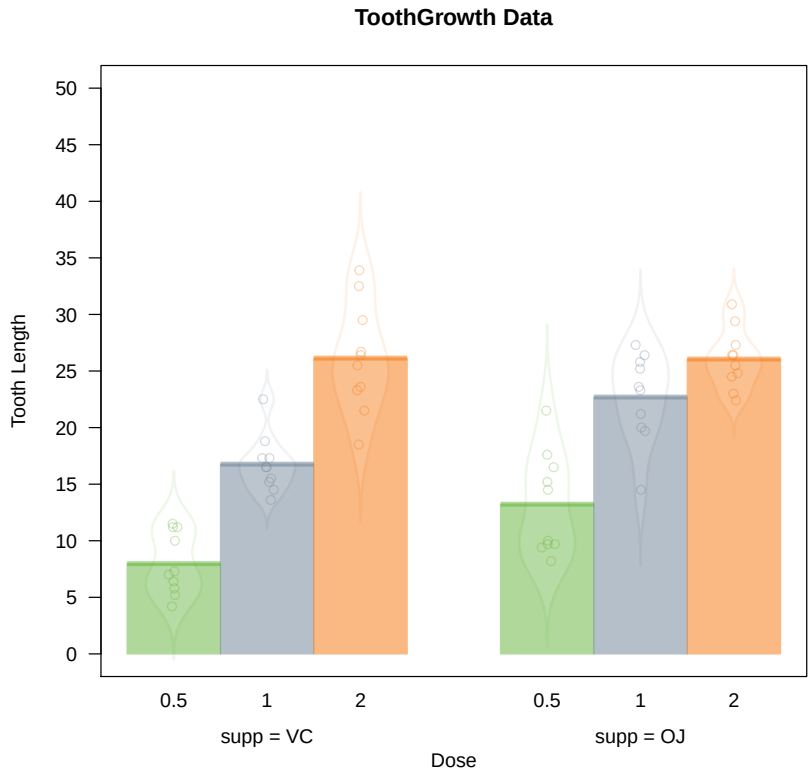
To see a specific palette in detail, run the following (replacing the name of the palette with the one you want)

```
piratepal(palette = "nemo", # nemo palette
          action = "show")
```



argument to the `piratepal()` function to create a souped up clustered barplot. For example, the `ToothGrowth` dataframe measures the length of Guinea Pig's teeth based on different supplements and different doses. We can plot both as follows:

```
# Plotting data from two IVs in pirateplot

pirateplot(formula = len ~ dose + supp,  # Two IVs
           data = ToothGrowth,
           xlab = "Dose",
           ylab = "Tooth Length",
           main = "ToothGrowth Data")
```



It's really easy to customize your pirateplots like crazy. In Figure 40 are four different pirateplots you can make of the ChickWeight data just by changing a few parameters in the `pirateplot()` function. To see how they were created, look at the help menu for pirateplot with `?pirateplot`:

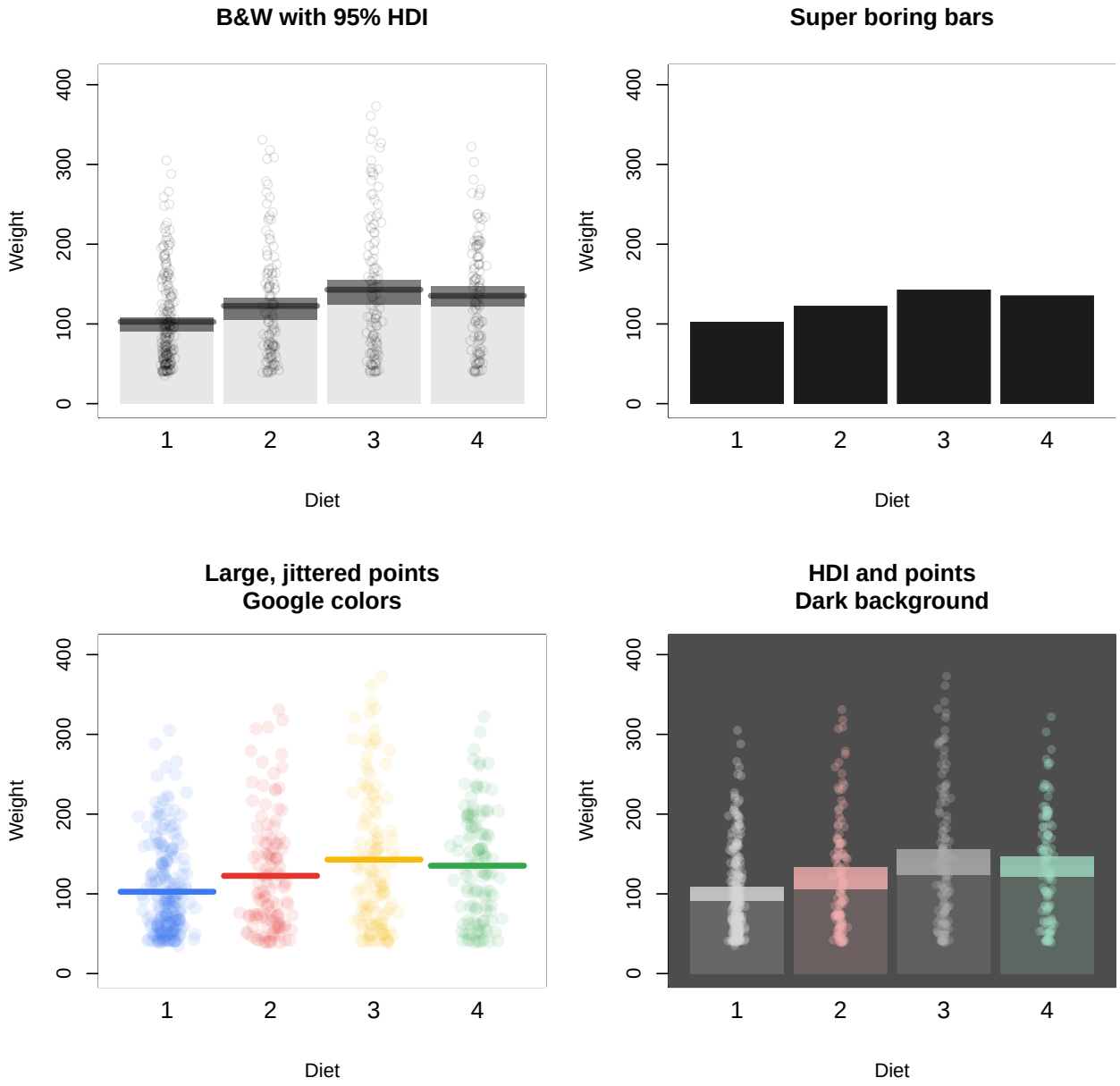


Figure 40: 4 different pirate plots of the ChickWeight dataset. Run `?pirateplot` to see the code for these examples.

Low-level plotting functions

Once you've created a plot with a high-level plotting function, you can add additional elements with *low-level* functions. You can add data points with `points()`, reference lines with `abline()`, text with `text()`, and legends with `legend()`.

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. 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.

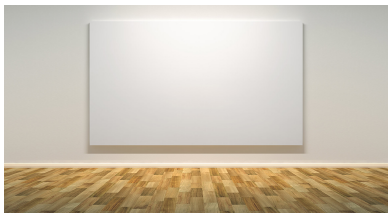
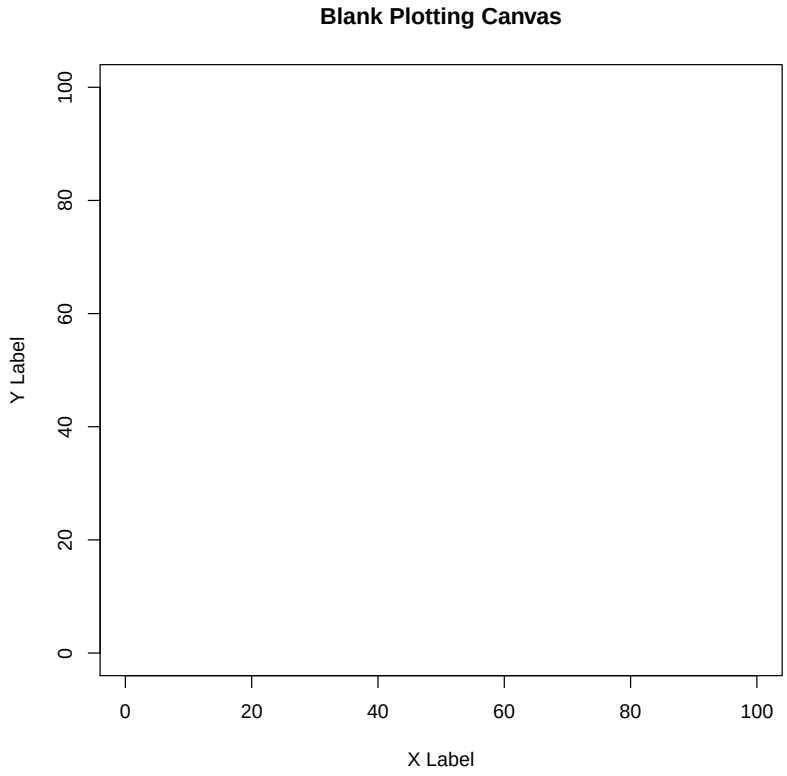


Figure 41: Ahhhh....a fresh blank canvas waiting for low-level plotting functions.

```
plot(x = 1,
     xlab = "X Label", ylab = "Y Label",
     xlim = c(0, 100), ylim = c(0, 100),
     main = "Blank Plotting Canvas",
     type = "n")
```



Adding new points to a plot with `points()`

To add new points to an existing plot, use the `points()` function[23].

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:

[23] 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()`

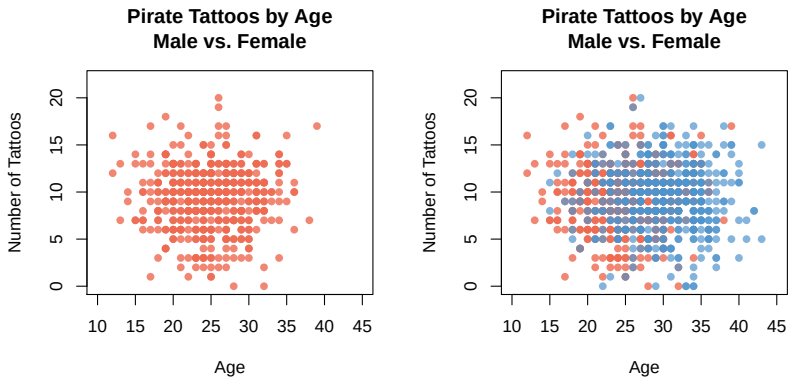
```
library(yarrr) # Load the yarrr package (for pirates dataset)

# Create the plot with male data
plot(x = pirates$age[pirates$sex == "male"],
     y = pirates$tattoos[pirates$sex == "male"],
     xlim = c(10, 45), ylim = c(0, 22),
     pch = 16,
     col = transparent("coral2", trans.val = .2),
     xlab = "Age", ylab = "Number of Tattoos",
     main = "Pirate Tattoos by Age\nMale vs. Female"
     )
```

Now, I'll add points for the female data with `points`:

```
# Add points for female data
points(x = pirates$age[pirates$sex == "female"],
       y = pirates$tattoos[pirates$sex == "female"],
       pch = 16,
       col = transparent("steelblue3", trans.val = .3)
       )
```

My first plotting command with `plot()` will create the left-hand figure below. The second plotting command with `points()` will add to the plot and create the right-hand figure below.

Because you can continue adding as many low-level plotting commands to a plot as you'd like, you can keep adding different types or colors of points by adding additional `points()` functions. However, keep in mind that because R plots each element on top of the previous one, early calls to `points()` might be covered by later calls. So add the points that you want in the foreground at the end!



Adding straight lines with `abline()`

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.

You can use `abline()` to add gridlines to a plot. To do this, enter the locations of horizontal lines with the `h` argument, and vertical lines with the `v` argument. The following code will create Figure 42:

```
# Create a blank plot.
plot(1, xlim = c(0, 100), ylim = c(0, 100), type = "n")

# Add horizontal and vertical gridlines at 0, 10, ... 100
abline(h = seq(from = 0, to = 100, by = 10),
       v = seq(from = 0, to = 100, by = 10))
```

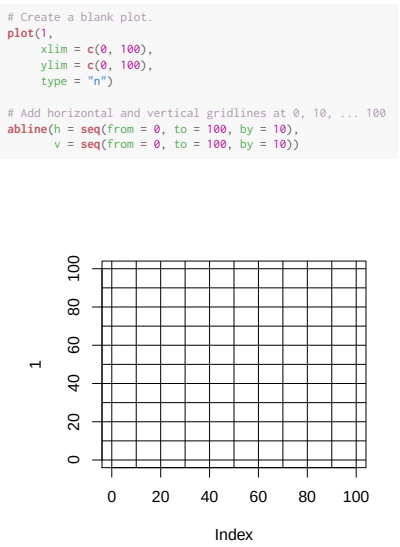


Figure 42: Adding gridlines with `abline()`

To change the look of your lines, use the `lty` argument, which changes the type of line (see Figure), `lwd`, which changes its thickness, and `col` which changes its color. The following code will use some of these arguments to create Figure 43:

```
# Create a blank plot.
plot(1, xlim = c(0, 100), ylim = c(0, 100), type = "n")

# Add horizontal and vertical gridlines at 0, 10, ... 100
abline(h = seq(from = 0, to = 100, by = 10),
       v = seq(from = 0, to = 100, by = 10),
       lty = 2,    # Dashed line
       col = gray(.5),    # Light gray color
       lwd = .5    # Thin line
       )
```

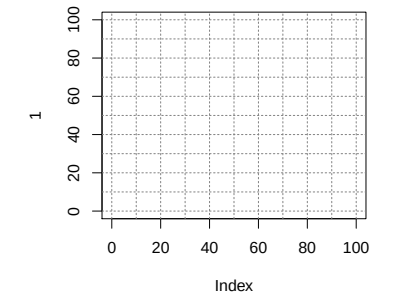


Figure 43: Adding thin, gray, dashed gridlines using the `lty`, `col`, and `lwd` arguments to `abline()`

The segments() function works very similarly to `abline()` – however, with the `segments()` function, you specify the beginning and end points of the segments. See the help menu (`?segments`) for details.

Once you've created a plot with gridlines using `abline()`, you can then add symbols with `points()`!

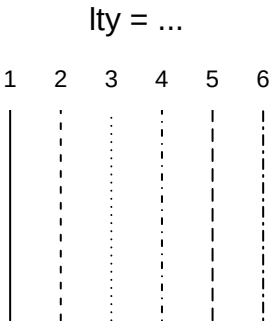


Figure 44: Line types generated from arguments to lty.

Adding text to a plot with `text()`

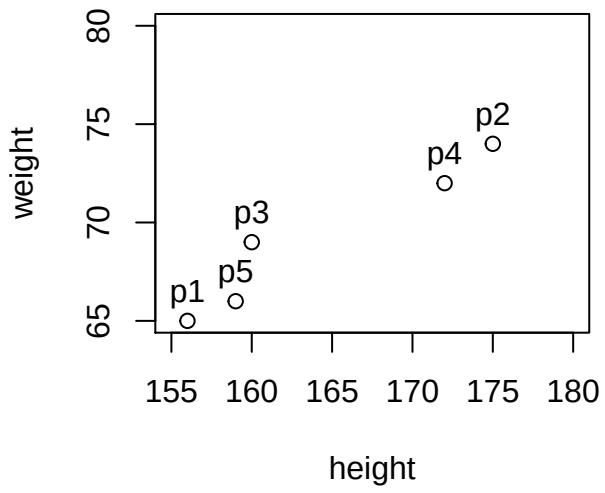
With `text()`, you can add text to a plot. You can use text() to highlight specific points of interest in the plot, or to add information (like a third variable) for every point in a plot. I've highlighted some of the key arguments to `text()` in Figure 45

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 46 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")
     )
```

You can do some cool things with `text()`. I'll create a scatterplot of data, and add data labels above each point:

```
height <- c(156, 175, 160, 172, 159)
weight <- c(65, 74, 69, 72, 66)
id <- c("p1", "p2", "p3", "p4", "p5")

plot(x = height, y = weight,
     xlim = c(155, 180), ylim = c(65, 80))

text(x = height, y = weight,
     labels = id, pos = 3)
```



Main arguments to `text()`:

- x, y: The location(s) of the text
- labels: The text to plot
- cex: The size of the text
- adj: How should the text be justified? 0 = left justified, .5 = centered, 1 = right justified
- pos: The position of the text relative to the x and y coordinates. Values of 1, 2, 3, and 4 put the text below, to the left, above, and to the right of the x-y coordinates respectively.

Figure 45: Main arguments to the `text()` function


```
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")
     )
```

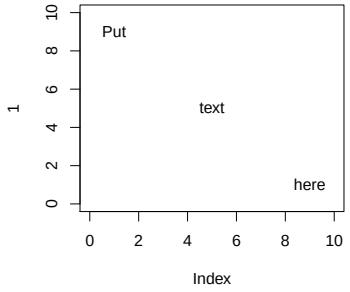



Figure 46: Adding text to a plot. The characters in the argument to `labels` will be plotted at the coordinates indicated by x and y.

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 47 for an example.

To plot text on separate lines in a plot, put the tag `"\n"` between lines.

```
plot(1, type = "n", main = "The \\n tag",
     xlab = "", ylab = "")

# Text without \n breaks
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"
     )
abline(h = 1, lty = 2)
# Text with \n 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"
     )
```

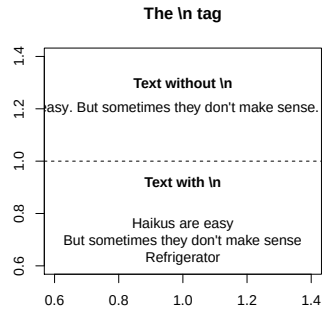


Figure 47: Using the `"\n"` tag to plot text on separate lines.

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()`:

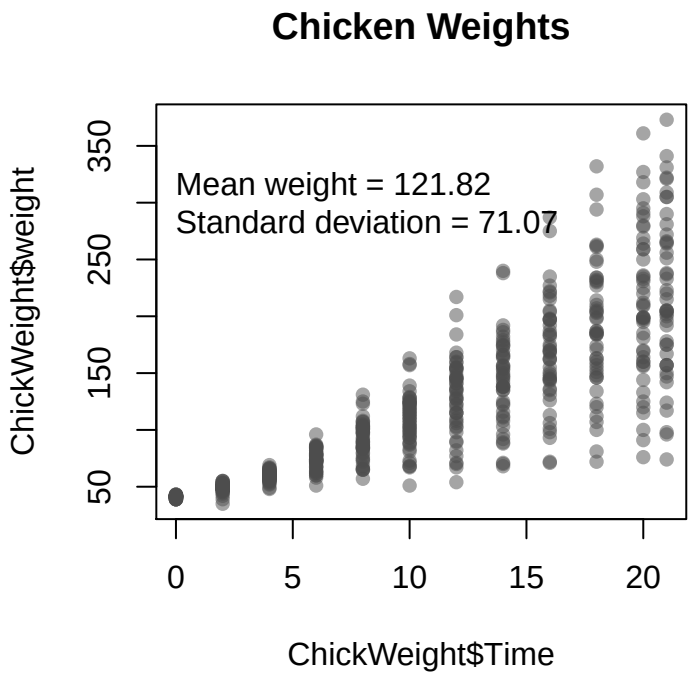
```
data <- ChickWeight$weight
mean(data)

## [1] 121.8183

paste("The mean of data is", mean(data)) # No rounding

## [1] "The mean of data is 121.818339100346"

paste("The mean of data is", round(mean(data), 2)) # No rounding

## [1] "The mean of data is 121.82"
```

When you include descriptive statistics in a plot, you will almost always want to use the `round(x, digits)` function to reduce the number of digits in the statistic.

Let's create a plot with labels using the `paste()` function. We'll plot the chicken weights over time, and add text to the plot specifying the overall mean and standard deviations of weights.

```
# Create the plot
plot(x = ChickWeight$Time,
     y = ChickWeight$weight,
     col = gray(.3, .5),
     pch = 16,
     main = "Chicken Weights")

# Add text
```

```
text(x = 0,
     y = 300,
     labels = paste("Mean weight = ",
                    round(mean(ChickWeight$weight), 2),
                    "\nStandard deviation = ",
                    round(sd(ChickWeight$weight), 2),
                    sep = ""),
     adj = 0)
```



curve()

The curve() function allows you to add a line showing a specific function or equation to a plot

curve()

`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 arguments such as `lty`, `col`, `lwd`, ...

For example, to add the function x^2 to a plot from the x-values -10 to 10, you can run the code:

```
curve(expr = x^2, from = -10, to = 10)
```

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:

```
my.fun <- function(x) {dnorm(x, mean = 10, sd = 3)}
curve(expr = my.fun, from = -10, to 10)
```

In Figure 48, I use the `curve()` function to create curves of several mathematical formulas.

```
plot(1, xlim = c(-5, 5), ylim = c(-5, 5),
     type = "n", main = "Plotting function lines with curve()",
     ylab = "", xlab = "")
abline(h = 0)
abline(v = 0)

require("RColorBrewer")
col.vec <- brewer.pal(12, name = "Set3")[4:7]

curve(expr = x^2, from = -5, to = 5,
      add = T, lwd = 2, col = col.vec[1])
curve(expr = x^.5, from = 0, to = 5,
      add = T, lwd = 2, col = col.vec[2])
curve(expr = sin, from = -5, to = 5,
      add = T, lwd = 2, col = col.vec[3])

my.fun <- function(x) {return(dnorm(x, mean = 2, sd = .2))}
curve(expr = my.fun, from = -5, to = 5,
      add = T, lwd = 2, col = col.vec[4])

legend("bottomright",
       legend = c("x^2", "x^.5", "sin(x)", "dnorm(x, 2, .2"),
       col = col.vec[1:4], lwd = 2,
       lty = 1, cex = .8, bty = "n"
       )
```

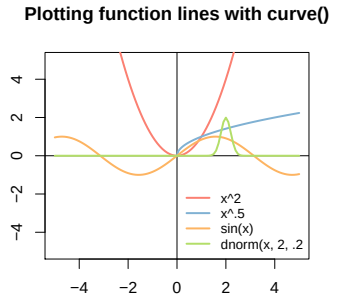


Figure 48: Using `curve()` to easily create lines of functions

legend()

The last low-level plotting function that we'll go over in detail is `legend()` which adds a legend to a plot. This function has the following arguments

`legend()`

`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

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:

```
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) # Point types
       )
```

In margin Figure I use this code to add a legend to plot containing data from males and females.

```
# Generate some random data
female.x <- rnorm(100)
female.y <- female.x + rnorm(100)
male.x <- rnorm(100)
male.y <- male.x + rnorm(100)

# Create plot with data from females
plot(female.x, female.y, pch = 16, col = 'blue',
     xlab = "x", ylab = "y", main = "Adding a legend with legend()"
     )

# Add data from males
points(male.x, male.y, pch = 16, col = 'orange')

# Add legend
legend("bottomright",
       legend = c("Females", "Males"),
       col = c('blue', 'orange'),
       pch = c(16, 16),
       bg = "white"
       )
```

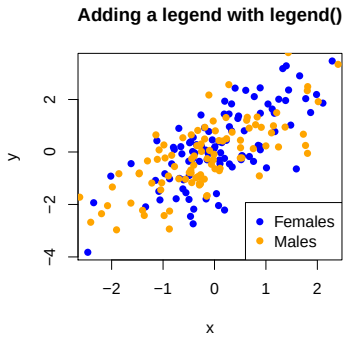


Figure 49: Creating a legend labeling the symbol types from different groups

Additional low-level plotting functions

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.

Additional low-level plotting functions

`rect()`

Add rectangles to a plot at coordinates specified by `xleft, ybottom, xright, ybottom`. For example, to add a rectangle with corners at (0, 0) and c(10, 10), specify `xleft = 0, ybottom = 0, xright = 10, ytop = 10`. Additional arguments like `col, border` change the color of the rectangle.

`polygon()`

Add a polygon to a plot at coordinates specified by vectors x and y. Additional arguments such as `col, border` change the color of the inside and border of the polygon

`segments(), arrows()`

Add segments (lines with fixed endings), or arrows to a plot.

`symbols(add = T)`

Add symbols (circles, squares, rectangles, stars, thermometers) to a plot. The dimensions of each symbol are specified with specific input types. See `?symbols` for details. Specify `add = T` to add to an existing plot or `add = F` to create a new plot.

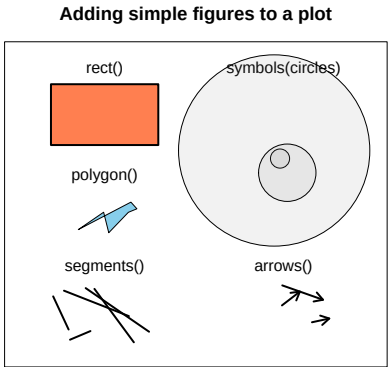
`axis()`

Add an additional axis to a plot (or add fully customizable x and y axes). Usually you only use this if you set `xaxt = "n", yaxt = "n"` in the original high-level plotting function.

`mtext()`

Add text to the margins of a plot. Look at the help menu for `mtext()` to see parameters for this function.

```
par(mar = c(0, 0, 3, 0))

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"
        )


# polygon(x = c(15, 35, 25, 15),
#         y = c(40, 40, 55, 40),
#         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)")

symbols(x = runif(3, 60, 90),
        y = runif(3, 60, 70),
        circles = c(1, .1, .3),
        add = T, 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
         )
```



Saving plots to a file

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()` or `jpeg()` functions. These functions will save your plot to either a .pdf of jpeg file.

`pdf()` and `jpeg()`

`file`

The name and file destination 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.

`family()`

An optional name of the font family to use for the plot. For example, `family = "Helvetica"` will use the Helvetica font for all text (assuming you have Helvetica on your system). For more help on using different fonts, look at section "Using extra fonts in R" in Chapter XX

`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 below).

To use these functions to save files, you need to follow 3 steps

1. Execute the `pdf()` or `jpeg()` functions with `file, width` and height arguments.
2. Execute all your plotting code.
3. Complete the file by executing the command `dev.off()`. This tells R that you're done creating the file.

Here's an example of the three steps.

```
# Step 1: Call the pdf command
pdf(file = "/figures/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
plot(1:10, 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()
```

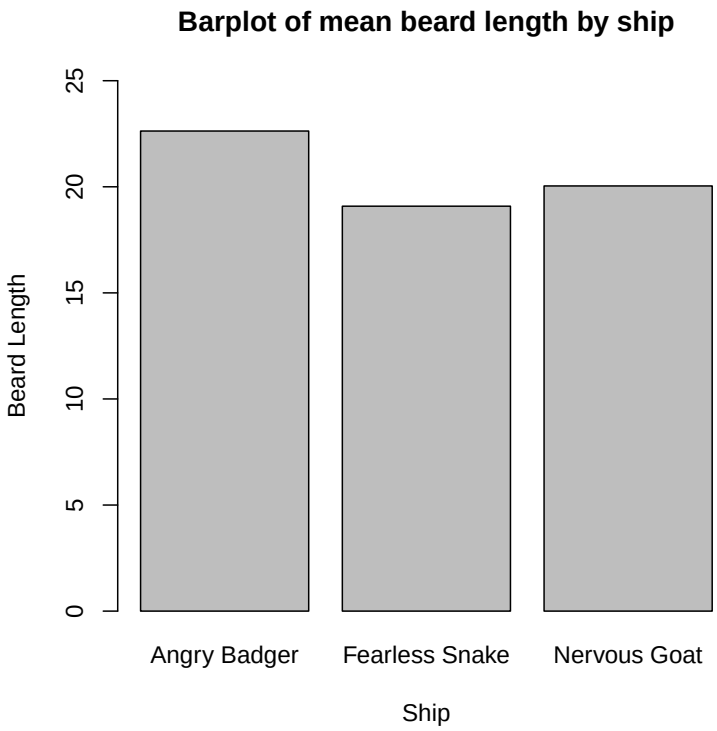
You'll notice that after you close the plot with `dev.off()`, you'll see a message in the prompt like "null device".

Using the command `pdf()` will save the file as a pdf. If you use `jpeg()`, it will be saved as a jpeg.

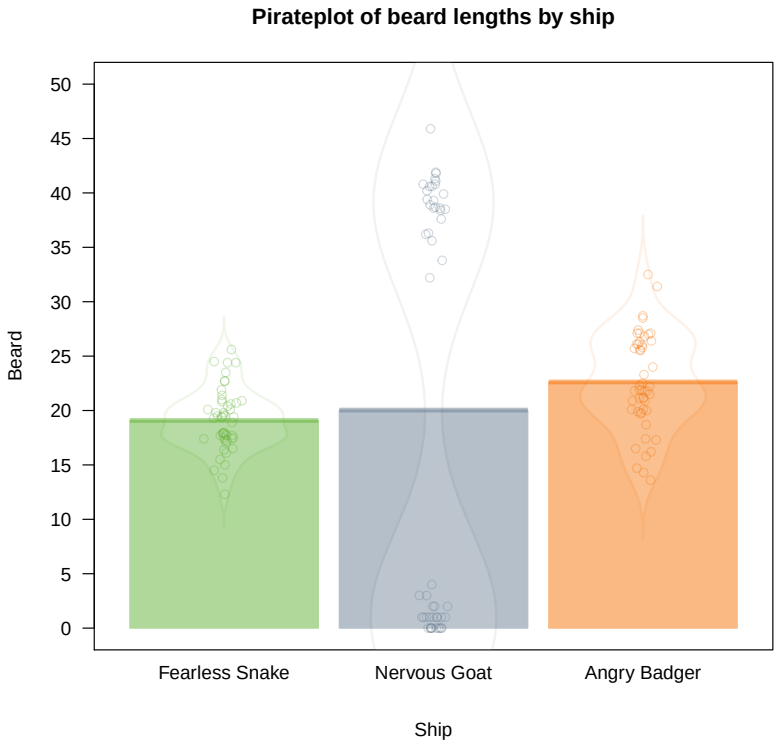
Test your R Might! Purdy pictures

For the following exercises, you'll use datasets from the `yarrr` package. Make sure to install and load the package

1. The `BeardLengths` dataframe 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:

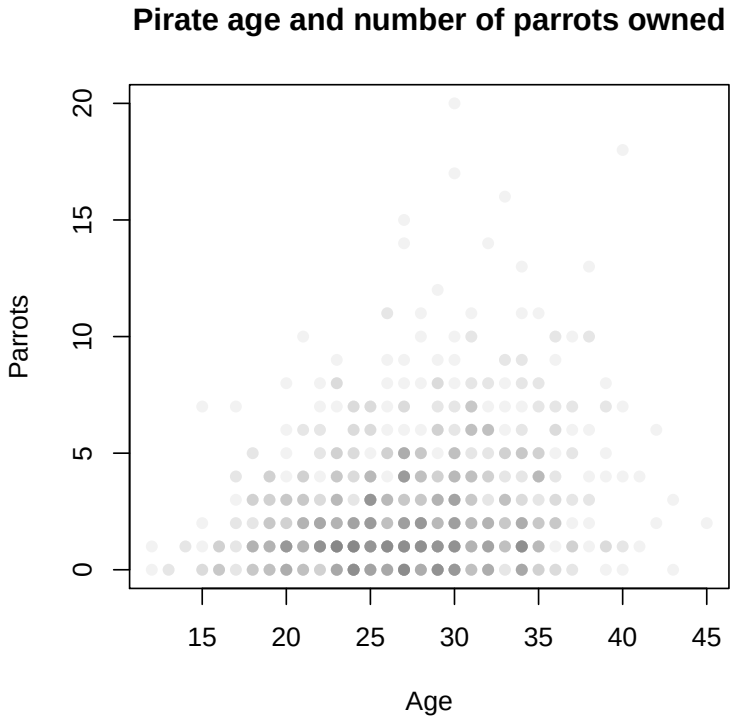


2. Now using the entire `BeardLengths` dataframe, create the following pirateplot:



3.

4. 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)).


Pirate age and number of parrots owned
Parrots
0
5
10
15
20
15
20
25
30
35
40
45
Age

10: Plotting: Part Deux

Advanced colors

Shades of gray with `gray()`

If you're a lonely, sexually repressed, 50+ year old housewife, then you might want to stick with shades of gray. If so, the function `gray(x)` is your answer. `gray()` is a function that takes a number (or vector of numbers) between 0 and 1 as an argument, and returns a shade of gray (or many shades of gray with a vector input). A value of 1 is equivalent to `"white"` while 0 is equivalent to `"black"`. This function is very helpful if you want to create shades of gray depending on the value of a numeric vector. For example, if you had survey data and plotted income on the x-axis and happiness on the y-axis of a scatterplot, you could determine the darkness of each point as a function of a third quantitative variable (such as number of children or amount of travel time to work). I plotted an example of this in Figure 50.

```
inc <- rnorm(n = 200, mean = 50, sd = 10)
hap <- inc + rnorm(n = 200, mean = 0, sd = 15)
drive <- inc + rnorm(n = 200, mean = 0, sd = 5)

plot(x = inc, y = hap, pch = 16,
     col = gray((drive - min(drive)) / max(drive - min(drive)), alpha = .4),
     cex = 1.5,
     xlab = "income", ylab = "happiness"
     )
```

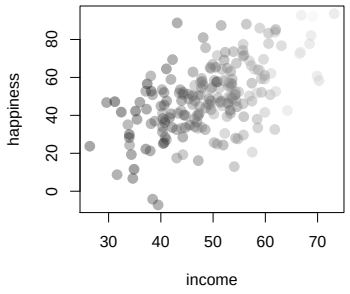
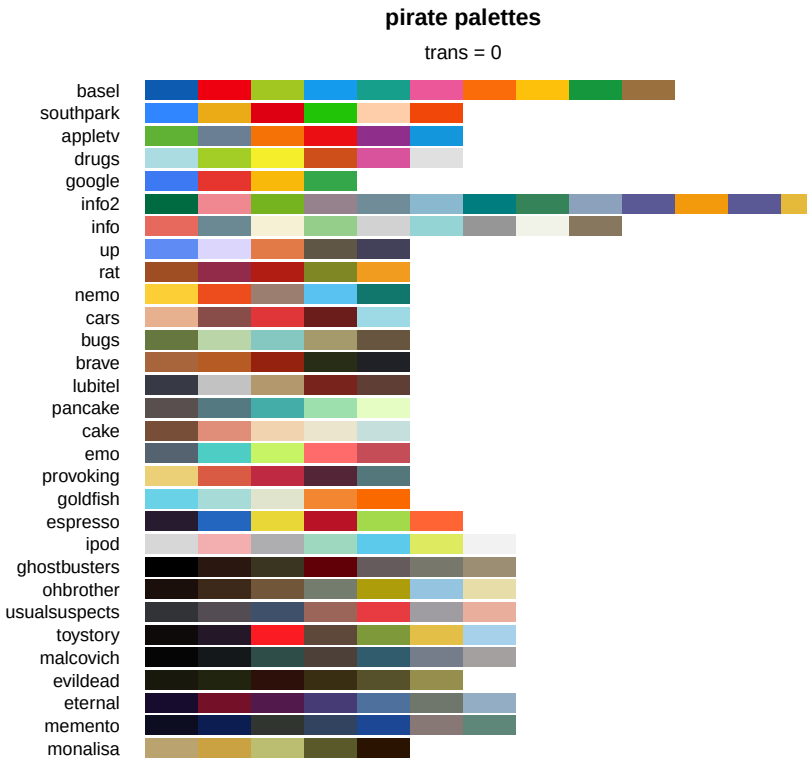


Figure 50: Using the `gray()` function to easily create shades of gray in plotting symbols based on numerical data.

Pirate Palettes

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 the following code:

```
library("yarrr")
piratepal(palette = "all",
          action = "show")
```



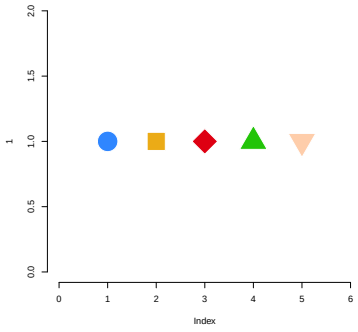
Once you find a color palette you like, you can save the colors as a vector by setting the `action` argument to `"return"`, and assigning the result to an object. For example, if I want to use the southpark palette and use them in a plot, I would do the following:

```
# Save the South Park palette
sp.cols <- piratepal(palette = "southpark", action = "return")

plot(x = 1:5, y = rep(1, 5),
     pch = c(21, 22, 23, 24, 25),
     main = "South Park Colors",
     bg = sp.cols, # Use the South Park Colors
     col = "white", cex = 3)
```

If you see a palette you like, you can see the colors (and their inspiration), in more detail as follows:

```
library("yarrr")
piratepal(palette = "basel",
          action = "show"
          )
```


```
# Save the South Park palette
sp.cols <- piratepal(palette = "southpark",
                     action = "return")

# Create a blank plot
plot(1, xlim = c(0, 6), ylim = c(0, 2),
     bty = "n", type = "n")

# Add points
points(x = 1:5, y = rep(1, 5),
     pch = c(21, 22, 23, 24, 25),
     bg = sp.cols, # Use the South Park Colors
     col = "white",
     cex = 5)
```

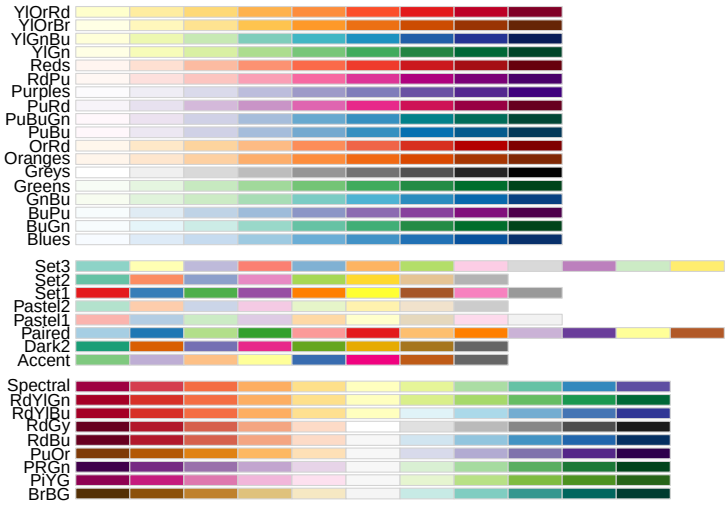


Color Palettes with the `RColorBrewer` *package*

If you use many colors in the same plot, it's probably a good idea to choose colors that compliment each other. An easy way to select colors that go well together is to use a *color palette* - a collection of colors known to go well together.

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:

```
require("RColorBrewer")
display.brewer.all()
```



To use one of the palettes, execute the function `brewer.pal(n, name)`, where `n` is the number of colors you want, and `name` is the name of the palette. For example, to get 4 colors from the color set "Set1", you'd use the code

```
my.colors <- brewer.pal(4, "Set1") # 4 colors from Set1
my.colors

## [1] "#E41A1C" "#377EB8" "#4DAF4A" "#984EA3"
```

I know the results look like gibberish, but trust me, R will interpret them as the colors in the palette. Once you store the output of the `brewer.pal()` function as a vector (something like `my.colors`),

you can then use this vector as an argument for the colors in your plot.

Numerically defined color gradients with `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.

`colorRamp2` allows you to do exactly this. The function has three arguments:

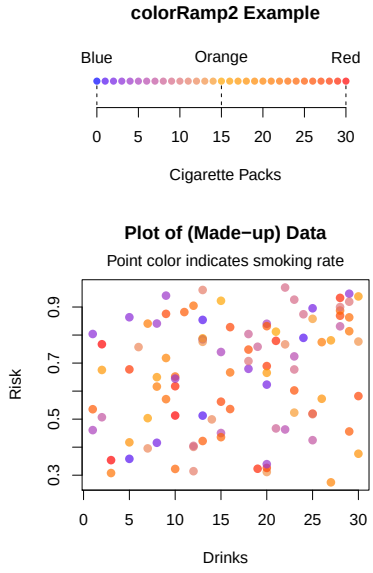
- `breaks`: A vector indicating the break points
- `colors`: A vector of colors corresponding to each value in `breaks`
- `transparency`: A value between 0 and 1 indicating the transparency (1 means fully transparent)

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:

```
smoking.colors <- colorRamp2(breaks = c(0, 15, 30),
                  colors = c("blue", "orange", "red"),
                  transparency = .3
                  )
```

```
library("RColorBrewer")
library("circlize")

# Create Data
drinks <- sample(1:30, size = 100, replace = T)
smokes <- sample(1:30, size = 100, replace = T)
risk <- 1 / (1 + exp(-drinks / 20 + rnorm(100, mean = 0, sd = 1)))

# Create color function from colorRamp2
smoking.colors <- colorRamp2(breaks = c(0, 15, 30),
                  colors = c("blue", "orange", "red"),
                  transparency = .3
                  )

# Set up plot layout
layout(mat = matrix(c(1, 2), nrow = 2, ncol = 1),
       heights = c(2.5, 5), widths = 4)

# Top Plot
par(mar = c(4, 4, 2, 1))
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", "Orange", "Red"))

# 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)
```



```
smoking.colors(0) # Equivalent to blue

## [1] "#0000FFB2"

smoking.colors(20) # Mix of orange and red

## [1] "#FF8200B2"
```

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.

Stealing any color from your screen 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 this method, I figured out the four colors of Google! Check out the margin Figure for the grand result.

```
google.colors <- c(
  rgb(19, 72, 206, maxColorValue = 255),
  rgb(206, 45, 35, maxColorValue = 255),
  rgb(253, 172, 10, maxColorValue = 255),
  rgb(18, 140, 70, maxColorValue = 255))

par(mar = rep(0, 4))

plot(1, xlim = c(0, 7), ylim = c(0, 1),
     xlab = "", ylab = "", xaxt = "n", yaxt = "n",
     type = "n", bty = "n"
     )

points(1:6, rep(.5, 6),
       pch = c(15, 16, 16, 17, 18, 15),
       col = google.colors[c(1, 2, 3, 1, 4, 2)],
       cex = 2.5)

text(3.5, .7, "Look familiar?", cex = 1.5)
```

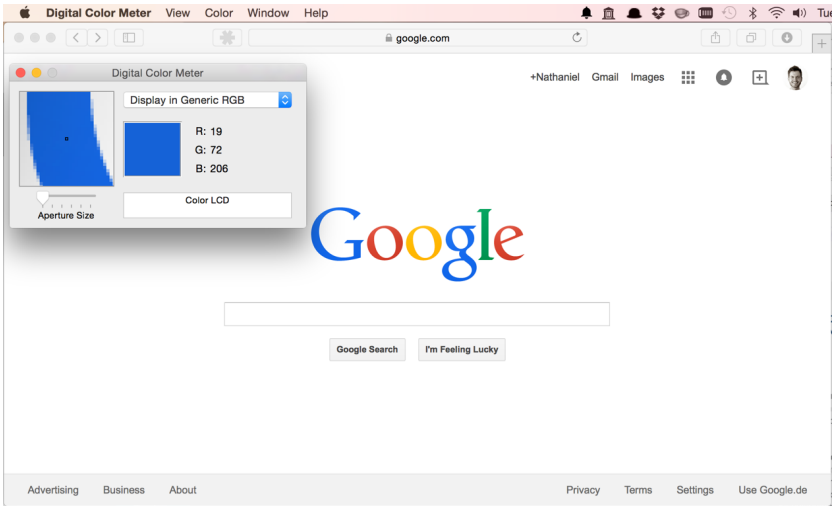


Figure 51: Stealing colors from the internet. Not illegal (yet).

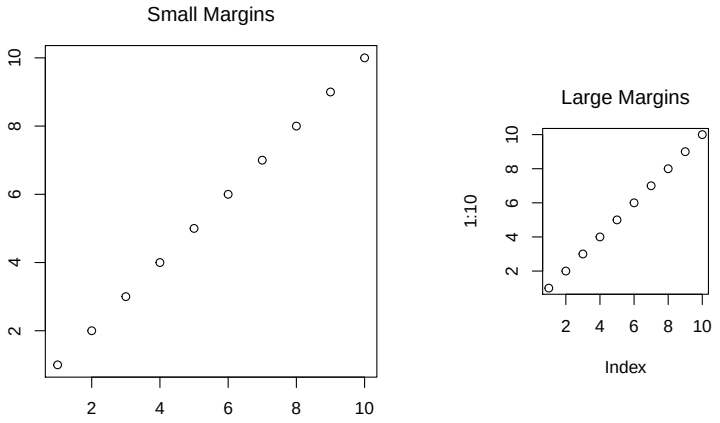
Plot margins

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 .

You can adjust the size of the margins by specifying a margin parameter using the syntax `par(mar = c(a, b, c, d))` before you execute your first high-level plotting function, where a, b, c and d are the size of the margins on the bottom, left, top, and right of 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.

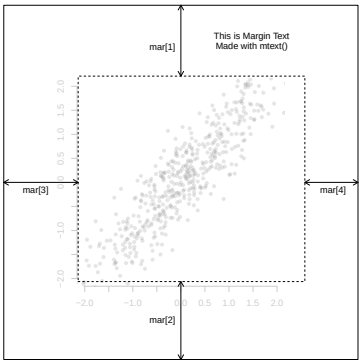
```
par(mfrow = c(1, 2)) # Put plots next to each other

# First Plot
par(mar = rep(2, 4)) # Set the margin on all sides to 2
plot(1:10)
mtext("Small Margins", side = 3, line = 1, cex = 1.2)

# Second Plot
par(mar = rep(6, 4)) # Set the margin on all sides to 6
plot(1:10)
mtext("Large Margins", side = 3, line = 1, cex = 1.2)
```



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.

```
par(mar = rep(8, 4))

x.vals <- rnorm(500)
y.vals <- x.vals + rnorm(500, sd = .5)

plot(x.vals, y.vals, xlim = c(-2, 2), ylim = c(-2, 2),
     main = "", xlab = "", ylab = "", xaxt = "n",
     yaxt = "n", bty = "n", pch = 16, col = gray(.5, alpha = .2))

axis(1, at = seq(-2, 2, .5), col.axis = gray(.8), col = gray(.8))
axis(2, at = seq(-2, 2, .5), col.axis = gray(.8), col = gray(.8))

par(new = T)
par(mar = rep(0, 4))
plot(1, xlim = c(0, 1), ylim = c(0, 1), type ="n",
     main = "", bty = "n", xlab = "", ylab = "", xaxt = "n", yaxt = "n")

rect(0, 0, 1, 1)

rect(.21, .22, .85, .8, lty = 2)

arrows(c(.5, .5, 0, .85),
       c(.8, .22, .5, .5),
       c(.5, .5, .21, 1),
       c(1, 0, .5, .5),
       code = 3, length = .1
       )

text(c(.5, .5, .09, .93),
     c(.88, .11, .5, .5),
     labels = c("mar[1]", "mar[2]", "mar[3]", "mar[4]"),
     pos = c(2, 2, 1, 1)
     )

text(.7, .9, "This is Margin Text\nMade with mtext()")
```



The default value for `mar` is `c(5.1, 4.1, 4.1, 2.1)`

Arranging multiple 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:

Simple plot layouts with par(mfrow) and par(mfcol)

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(3, 3)) # Create a 3 x 3 plotting matrix
```

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:

```
par(mfrow = c(1, 1))
```

If you don't reset the `mfrow` parameter, R will continue creating new plotting matrices.

Complex plot layouts with `layout()`

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. Let's go through the main arguments of `layout()`:

```
layout(mat, widths, heights)
```

```
par(mfrow = c(3, 3))
par(mar = rep(2.5, 4))

for(i in 1:9) { # Loop across plots

# Generate data
x <- rnorm(100)
y <- x + rnorm(100)

# Plot data
  plot(x, y, xlim = c(-2, 2), ylim = c(-2, 2),
       col.main = "gray",
       pch = 16, col = gray(.0, alpha = .1),
       xaxt = "n", yaxt = "n"
       )

# Add a regression line for fun
abline(lm(y ~ x), col = "gray", lty = 2)


# Add gray axes
axis(1, col.axis = "gray",
     col.lab = gray(.1), col = "gray")

axis(2, col.axis = "gray",
     col.lab = gray(.1), col = "gray")

# Add large index text
text(0, 0, i, cex = 7)

# Create box around border
box(which = "figure", lty = 2)

}
```

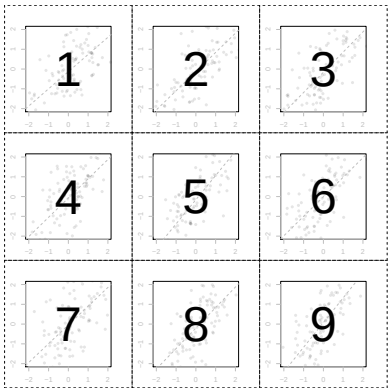


Figure 52: A matrix of plotting regions created by `par(mfrow = c(3, 3))`

- `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.

The `layout()` 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 (see margin Figure 53)

Now we're ready to put the plots together

```
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, 1) # Widths of the two columns
       )

x.vals <- rnorm(100, mean = 100, sd = 10)
y.vals <- x.vals + rnorm(100, mean = 0, sd = 10)

# Plot 1: Scatterplot
par(mar = c(5, 4, 0, 0))
plot(x.vals, y.vals)
```

```
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)
```

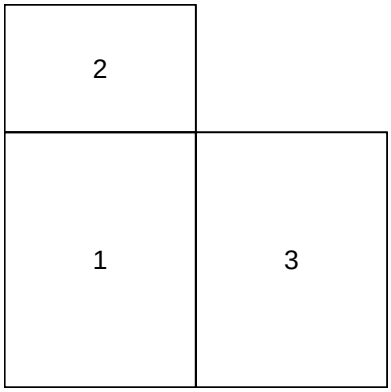
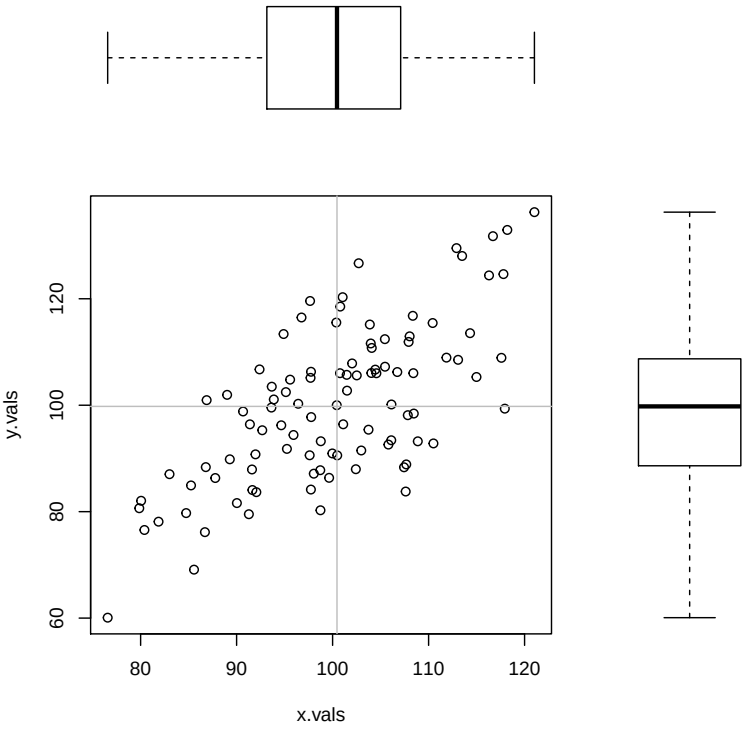


Figure 53: A plotting layout created by setting a layout matrix and specific heights and widths.

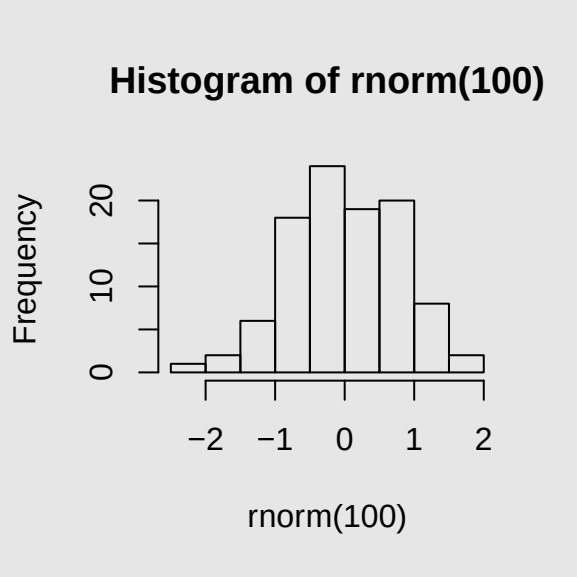
```r
abline(h = median(y.vals), lty = 1, col = "gray")
abline(v = median(x.vals), lty = 1, col = "gray")


# Plot 2: X boxplot
par(mar = c(0, 4, 0, 0))
boxplot(x.vals, xaxt = "n",
        yaxt = "n", bty = "n", yaxt = "n",
        col = "white", frame = F, horizontal = T)

# Plot 3: Y boxplot
par(mar = c(5, 0, 0, 0))
boxplot(y.vals, xaxt = "n",
        yaxt = "n", bty = "n", yaxt = "n",
        col = "white", frame = F)
```



Additional Tips

- To change the background color of a plot, add the command `par(bg = mycolor)` (where my.color 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))
hist(x = rnorm(100))
```



See Figure 54 for a nicer example.

- Sometimes you'll mess so much with plotting parameters that you may want to set things back to their default value. To see the default values for all the plotting parameters, execute the code `par()` to print the default parameter values for all plotting parameters to the console.

```
pdf("/Users/nphillips/Dropbox/Git/YaRrr_Book/media/parrothelvetica.pdf",
    width = 8, height = 6)

parrot.data <- data.frame(
  "parrots" = 0:6,
  "female" = c(200, 150, 100, 175, 55, 25, 10),
  "male" = c(150, 125, 180, 242, 10, 62, 5)
  )

n.data <- nrow(parrot.data)

par(bg = rgb(61, 55, 72, maxColorValue = 255),
    mar = c(8, 6, 6, 3)
    )

plot(1, xlab = "", ylab = "", xaxt = "n",
     yaxt = "n", main = "", bty = "n", type = "n",
     ylim = c(0, 250), xlim = c(.5, n.data + .5)
     )

abline(h = seq(0, 250, 50), lty = 3, col = gray(.95), lwd = 1)

mtext(text = seq(50, 250, 50),
      side = 2, at = seq(50, 250, 50),
      las = 1, line = 1, col = gray(.95))

mtext(text = paste(0:(n.data - 1), " Parrots"),
      side = 1, at = 1:n.data, las = 1,
      line = 1, col = gray(.95))


female.col <- gray(1, alpha = .7)
male.col <- rgb (226, 89, 92, maxColorValue = 255, alpha = 220)

rect.width <- .35
rect.space <- .04

rect(1:n.data - rect.width - rect.space / 2,
     rep(0, n.data),
     1:n.data - rect.space / 2,
     parrot.data$female,
     col = female.col, border = NA
     )

rect(1:n.data + rect.space / 2,
     rep(0, n.data),
     1:n.data + rect.width + rect.space / 2,
     parrot.data$male,
     col = male.col, border = NA
     )

legend(n.data - 1, 250, c("Male Pirates", "Female Pirates"),
       col = c(female.col, male.col), pch = rep(15, 2),
       bty = "n", pt.cex = 1.5, text.col = "white"
       )

mtext("Number of parrots owned by pirates", side = 3,
      at = n.data + .5, adj = 1, cex = 1.2, col = "white")

mtext("Source: Drunken survey on 22 May 2015", side = 1,
      at = 0, adj = 0, line = 3, font = 3, col = "white")

dev.off()


## pdf
##   2


#embed_fonts("/Users/nphillips/Dropbox/Git/YaRrr_Book/media/parrothelvetica.pdf") #
```

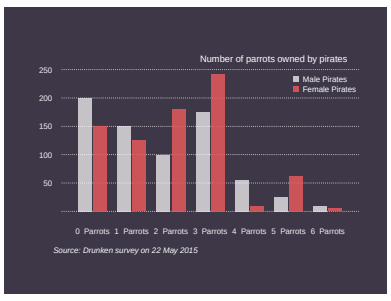


Figure 54: Use `par(bg = my.color)` before creating a plot to add a colored background. The design of this plot was inspired by `http://www.vox.com/2015/5/20/8625785/expensive-wine-`

11: Inferential Statistics: 1 and 2-sample Null-Hypothesis tests

Do we get more treasure from chests buried in the 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 cover three main concepts: Descriptive statistics, Test statistics, and p-values. To do this, let's talk about body piercings.

Null vs. Alternative Hypotheses, Descriptive Statistics, Test Statistics, and p-values: A very short introduction

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 55:

```
american.bp <- c(3, 5, 2, 1, 4, 4, 6, 3, 5, 4)
european.bp <- c(8, 5, 9, 7, 6, 8, 8, 6, 9, 7)
```

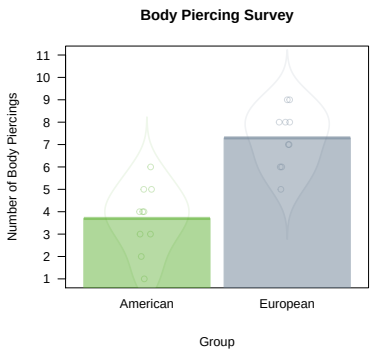


Figure 55: A Pirateplot (using the `pirateplot()` function in the `yarrr` package) of the body piercing data
.

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.

Null Hypothesis: Everything is the same

Alternative Hypothesis: Everything is *not* the same

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.

Descriptive Statistics

Descriptive 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:

```
summary(american.bp)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    3.00    4.00    3.70    4.75    6.00

summary(european.bp)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    5.00    6.25    7.50    7.30    8.00    9.00
```

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 7.3 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*

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.

```
bp.test <- t.test(x = american.bp,
                  y = european.bp,
                  alternative = "two.sided")
```

I can get the test statistic from my new `bp.test` object by using the $ operator as follows:

```
bp.test$statistic

##         t
## -5.676354
```

It looks like our test-statistic is -5.68. If there was really no difference between the groups of pirates, we would expect a test statistic close to 0. Because test-statistic is -5.68, 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.

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*.

> Assuming that there is no difference between the groups, what is the probability that we would have gotten a test statistic as extreme as the one we actually got?

For this problem, we can access the p-value as follows:

```
bp.test$p.value

## [1] 2.304418e-05
```

The p-value we got was .000022, 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 −5.68 is around 0.0022%. This is very small – in other words, it's close to impossible. Because our p-value is so small, we conclude that the two populations must *not* be the same.

Figure 56: 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.

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.

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.

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%.

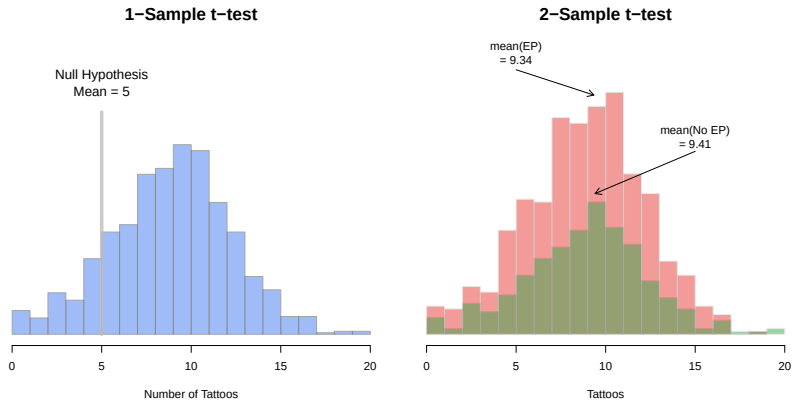
No. Your partner is a man. The probability that the null hypothesis is true (i.e. that he is not pregnant), is 100%, not 1%. Your stupid boyfriend doesn't understand basic biology and decided to buy an expensive pregnancy test anyway.

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.

Figure 57: 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!

T-test with `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`).



One-Sample t-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, just enter a vector of data as the main argument to `t.test()`, and put the hypothesized value as the argument to `mu`:

```
t.test(x = pirates$tattoos,  # Vector of data
       mu = 5)               # Null: Mean is 5

##
##  One Sample t-test
##
## data:  pirates$tattoos
## t = 40.735, df = 999, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 5
## 95 percent confidence interval:
##  9.15187 9.57213
## sample estimates:
## mean of x
```

Here are some optional arguments to `t.test()`

`alternative` A character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.

`mu` A number indicating the true value of the mean (or difference in means if you are performing a two sample test). The default is 0.

`paired` A logical indicating whether you want a paired t-test.

`var.equal` A logical variable indicating whether to treat the two variances as being equal.

`f.level` The confidence level of the interval.

`subset` An optional vector specifying a subset of observations to be used.

```
##     9.362
```

As you can see, the function printed lots of information: the descriptive mean of the sample was 9.36, the test statistic (40.74), and the p-value was $1.5546485 \times 10^{-214}$.

Now, what happens if I change the null hypothesis to a mean of 9.3? Because the descriptive sample mean was 9.36, quite close to 9.3, the test statistic should decrease, and the p-value should increase:

```
t.test(x = pirates$tattoos,
       mu = 9.3)  #Null: Mean is 9.3

##
##  One Sample t-test
##
## data:  pirates$tattoos
## t = 0.579, df = 999, p-value = 0.5627
## alternative hypothesis: true mean is not equal to 9.3
## 95 percent confidence interval:
##  9.15187 9.57213
## sample estimates:
## mean of x
##     9.362
```

Looks good. The test statistic decreased to just 0.58, and the p-value increased to 0.56. In other words, our sample mean of 9.36 is very consistent with the hypothesis that the true population mean is 9.30.

Two-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. For example, if someone said that pirates who wear eye patches have fewer tattoos on average than those who don't wear eye patches, we could test this with a two-sample t-test. To do a two-sample t-test, enter the dependent variable (e.g.; tattoos) and independent variable (e.g.; eye patch) as a formula in the form `dv ~ iv`, and specify the dataset containing the data in `data`:

```
t.test(formula = tattoos ~ eyepatch,
       data = pirates)

##
##  Welch Two Sample t-test
##
```

```
## data:  tattoos by eyepatch
## t = 0.29452, df = 621.27, p-value = 0.7685
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.3866154  0.5230411
## sample estimates:
## mean in group 0 mean in group 1
##        9.407975        9.339763
```

This test gave us a test statistic of 0.29 and a p-value of 0.77.

Saving and using hypothesis test objects

When you run a t-test using `t.test()`, you can save the entire object as a *hypothesis test* object using basic assignment. I'll repeat the previous test but save the result as a new object called `tattoo.eyepatch.test`

```
tattoo.eyepatch.test <- t.test(formula = tattoos ~ eyepatch,
                               data = pirates)
```

To see all the information contained in the test object, use the `names()` function

```
names(tattoo.eyepatch.test)

## [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
## [6] "null.value"  "alternative" "method"      "data.name"
```

Once you know the names of what's in the object, you can access specific values by name using the $ operator (just like in dataframes). In the code below, I'll access just the p-value and test statistic from my new `tattoo.eyepatch.test` object:

```
# What was the p-value?
tattoo.eyepatch.test$p.value

## [1] 0.7684595

# What was the test statistic?
tattoo.eyepatch.test$statistic

##         t
## 0.2945193
```

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)

## Error in t.test.formula(formula = tattoos ~ age, data =
pirates): grouping factor must have exactly 2 levels
```

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:

```
t.test(formula = tattoos ~ age,
       data = pirates,
       subset = age %in% c(29, 30))  # Compare age of 29 to 30

##
##  Welch Two Sample t-test
##
## data:  tattoos by age
## t = -1.149, df = 132.74, p-value = 0.2526
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.7207322  0.4562161
## sample estimates:
## mean in group 29 mean in group 30
##         8.967742         9.600000
```

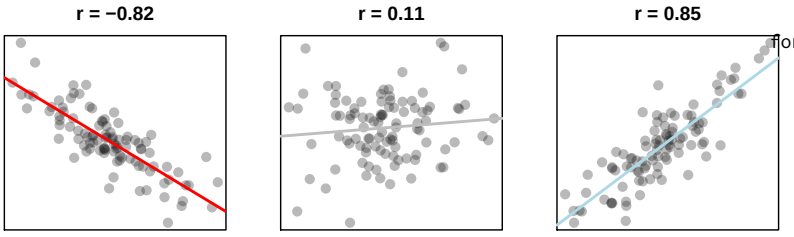
We got a p-value of 0.25 which is pretty high and suggests that we should fail to reject the null hypothesis.

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 handbands 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 = 1.5292, df = 480.25, p-value = 0.1269
## alternative hypothesis: true difference in means
## 95 percent confidence interval:
##  -0.1362564  1.0926772
## sample estimates:
##  mean in group CCCC mean in group JSSFP
##            9.610442            9.132231
```

Correlation test with `cor.test()`

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, use the `cor.test()` function. 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:

```
# Is there a correlation between a pirate's age and
#  the number of parrots (s)he's owned?

age.parrots.test <- cor.test(formula = ~ age + parrots,
                             data = pirates)
```

Now let's print the result:

```
age.parrots.test

##
##  Pearson's product-moment correlation
##
## data:  age and parrots
## t = 7.4153, df = 998, p-value = 2.588e-13
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.1689151 0.2864504
## sample estimates:
##       cor
## 0.2285152
```

Here are some optional arguments to `cor.test()`. As always, check the help menu with `?cor.test` for additional information and examples:

`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 indicating the direction of the test. 't' stands for two-sided, 'l' stands for less than, and 'g' stands for greater than.

`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.

`f.level` The confidence level of the interval.

`subset` A vector specifying a subset of observations to use.

Looks like we have a positive correlation of 0.23 and a very small p-value of $2.5884453 \times 10^{-13}$. To see what information we can extract for this test, let's run the command `names()` on the test object:

```
names(age.parrots.test)

## [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.test$conf.int

## [1] 0.1689151 0.2864504
## 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 = 4.0423, df = 489, p-value = 6.147e-05
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.09280611 0.26410932
## sample estimates:
##       cor
## 0.1798206
```

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.

Chi-square 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 $\chi^2 = 0$ which means no relationship.

Here are some optional arguments to `chisq.test()`. As always, check the help menu with `?chisq.test` for additional information and examples:

- `x` A contingency table of one (or two) categorical variables

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? We can test this on the pirates dataset by testing if there are an equal number of pirates from each college. Here is a table of the college data:

To get this table of the pirate college data, just use the `table()` function:

```
table(pirates$college)

##
##  CCCC JSSFP
##   667   333
```

college	frequency
CCCC	667
JSSFP	333

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. As the main argument to the `chisq.test()` function, you enter a *table* of the data using the `table()` function rather than the original vector.

Don't forget that you should enter your data into the `chisq.test()` as a table using the `table()` function!

```
# Are all colleges equally prevelant?

chisq.test(x = table(pirates$college))

##
##  Chi-squared test for given probabilities
##
## data:  table(pirates$college)
## X-squared = 111.56, df = 1, p-value < 2.2e-16
```

Indeed, with a test statistic of 111.56 and a p-value of $4.4699898 \times 10^{-26}$, we can safely reject the null hypothesis and conclude that certain college *are* more popular than others.

2-sample chi-square test

You can also conduct a 2-sample chi-square test by entering two separate vectors as arguments to `chisq.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:

```
table(pirates$eyepatch,
      pirates$favorite.pirate)
```

	Anicetus	Blackbeard	Edward Low	Hook	Jack Sparrow	Lewis Scot
CCCC	68	53	81	67	337	61
JSSFP	37	32	67	48	103	46

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?

chisq.test(x = table(pirates$college,
                     pirates$eyepatch))

##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  table(pirates$college, pirates$eyepatch)
## X-squared = 2.0729, df = 1, p-value = 0.1499
```

It looks like we got a test statistic of $\chi^2 = 2.07$ and a p-value of 0.15. 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.

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 = 1.3542, df = 116.92, p-value = 0.1783
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.3834475  2.0416673
## sample estimates:
##  mean in group no mean in group yes
##          28.14286          27.31375
```

It looks like the test statistic is 1.35, degrees of freedom is 116.92, and the p-value is 0.178. Let's see how the apa function gets these values directly from the test object:

```
library(yarrr) # Load the yarrr library
apa(test.result)

## [1] "mean difference = -0.83, t(116.92) = 1.35, p = 0.18 (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

apa(cor.test(formula = ~ age + parrots,
             data = pirates))

## [1] "r = 0.23, t(998) = 7.42, 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.

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.

12: ANOVA and Factorial Designs

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[24]. 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 *Mènage á trois 'NOVA*[25].

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.

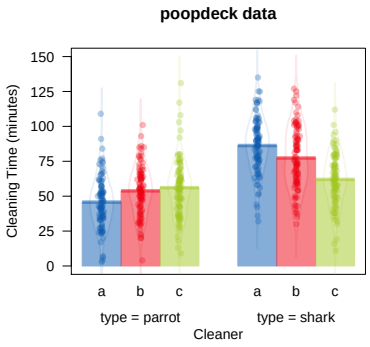
Thankfully, this experiment has already been conducted. The data are recorded in a dataframe called `poopdeck` in the yarrr package. Here's how the first 8 rows of the data look:

[24] 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)

[25] Ok maybe it's not yet, but we you repeat it enough it will be

We can visualise the poopdeck dataus-ing (of course) a pirate plot:

```
pirateplot(time ~ cleaner + type,
           data = poopdeck,
           ylim = c(0, 150),
           xlab = "Cleaner",
           ylab = "Cleaning Time (minutes)",
           main = "poopdeck data",
           point.pch = 16,
           pal = "basel",
           back.col = gray(.98)
           )
```



day	cleaner	type	time
1	a	parrot	47
1	b	parrot	55
1	c	parrot	64
1	a	shark	101
1	b	shark	76
1	c	shark	63
2	a	parrot	25
2	b	parrot	44

Given this data, we can use ANOVAs to answer three separate questions:

1. Is there a difference between the different cleaners on cleaning time (ignoring poop type)? (*One-way ANOVA*)
2. Is there a difference between the different poop types on cleaning time (ignoring which cleaner is used)? (*One-way ANOVA*)
3. Is there a *unique* effect of the cleaner or poop types on cleaning time? (*Two-way ANOVA*)
4. Does the effect of cleaner depend on the poop type? (*Interaction between cleaner and type*)

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[26]

For the rest of this chapter, I will refer to full-factorial between-subjects ANOVAs as 'standard' ANOVAs

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

[26] 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.

the means of groups. Specifically, it looks at how variable data are *within* groups and comapres 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 58 for a visual depiction of an ANOVA.

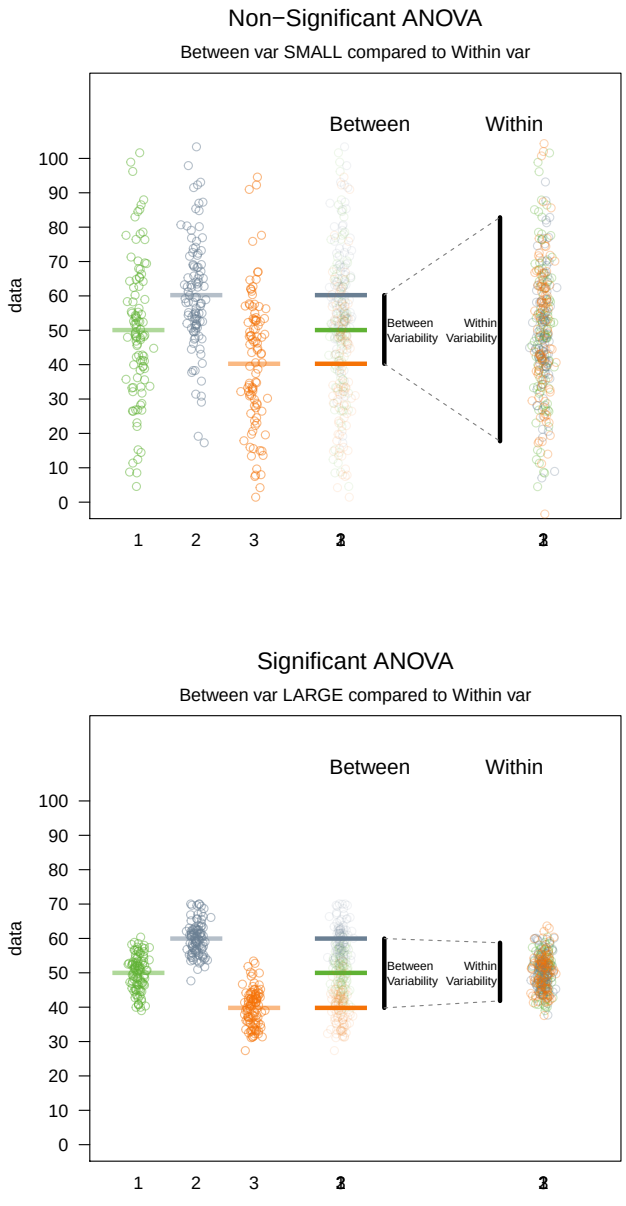


Figure 58: How ANOVAs work. ANOVA compares the variability *between* groups (i.e.; the differences in the means) to the variability *within* groups (i.e.; the differences between members within groups). If the variability between groups is *small* compared to the variability between groups, ANOVA will return a non-significant result – suggesting taht 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 that the groups *are* really different.

4 Steps to conduct a standard ANOVA in R

Here are the 4 steps you should follow to conduct a standard ANOVA in R:

Step 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 2 Create a summary ANOVA table by applying the `summary()` function to the ANOVA object you created in Step 1.

Step 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 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.

Let's do an example by running both a one-way and two-way ANOVA on the `poopdeck` data:

One-way (1 IV) ANOVA

You conduct a one-way ANOVA when you testing one factor independent variable and are ignoring all other possible variables. Let's use the poopdeck data and do a one-way ANOVA with the cleaner type as the independent variable, and the cleaning time as the dependent variable.

Step 1: Create an ANOVA object from the regression object with `aov()`

First, we'll create an ANOVA object with `aov`. Because cleaner is the independent variable, and time is the dependent variable, we'll include the argument `formula = time ~ cleaner`

```
cleaner.aov <- aov(formula = time ~ cleaner,
                   data = poopdeck)
```

Step 2: Look at summary tables of the ANOVA with `summary()`

Now, to see a full ANOVA summary table of the ANOVA object, apply the `summary()` to the ANOVA object from Step 1.

```
summary(cleaner.aov)

##              Df Sum Sq Mean Sq F value  Pr(>F)
## cleaner       2   6057    3028   5.294 0.00526 **
## 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,w e need to conduct a post-hoc test.

Step 3: Do pairwise comparisons with `TukeyHSD()`

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[27]. One of the most common post-hoc tests for standard ANOVAs is the Tukey Honestly Significant Difference (HSD) test[28]. To do an HSD test, apply the `TukeyHSD()` function to your ANOVA object as follows:

[27] To learn more about the logic behind different post-hoc tests, check out the Wikipedia page here: Post-hoc Test Wikipedia

[28] To see additional information about the Tukey HSD test, check out the Wikipedia page here: Tukey HSD Wikipedia

```
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  -6.039575  5.1995746 0.9831441
## c-a -6.94 -12.559575 -1.3204254 0.0107324
## c-b -6.52 -12.139575 -0.9004254 0.0180906
```

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.

Step 4: Look at the coefficients in a regression analysis with `lm()`

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 indpendent 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

```
# 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)   66.020      1.691  39.037  < 2e-16 ***
## cleanerb      -0.420      2.392  -0.176  0.86066
## cleanerc      -6.940      2.392  -2.902  0.00385 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 23.92 on 597 degrees of freedom
## Multiple R-squared:  0.01743,Adjusted R-squared:  0.01413
## F-statistic: 5.294 on 2 and 597 DF,  p-value: 0.005261
```

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!

Multiple independent variables with + (`y ~ x1 + x2 + ...`)

To conduct a two-way ANOVA or a *Mènage á 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)
```

Here's the summary table:

```
# Step 2: Get ANOVA table with summary()
summary(cleaner.type.aov)

##              Df Sum Sq Mean Sq F value  Pr(>F)
## cleaner       2   6057    3028   6.945 0.00104 **
## type          1  81620   81620 187.177 < 2e-16 ***
## Residuals   596 259891     436
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Again, given significant effects, we can proceed with post-hoc tests:

```
# 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  -5.326395  4.486395 0.9779465
## c-a -6.94 -11.846395 -2.033605 0.0027112
## c-b -6.52 -11.426395 -1.613605 0.0053376
##
## $type
##                  diff      lwr      upr p adj
## shark-parrot 23.32667 19.97811 26.67522     0
```

It looks like we found significant effects of both independent variables. The only non-significant group difference we found is between cleaner b and cleaner a.

Interactions between variables with * *(*`y ~ x1 * x2`*)*

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[29].

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
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   7.824 0.000443 ***
## type           1  81620   81620 210.863  < 2e-16 ***
## cleaner:type   2  29968   14984  38.710  < 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)
```

Again, to interpret that nature of the results, it's helpful to repeat the analysis as a regression using the `lm()` function:

```
# Step 4: Look at regression coefficients
cleaner.type.lm <- lm(formula = time ~ cleaner + typ
                      data = poopdeck)

summary(cleaner.type.lm)
```

```
##
## Call:
## lm(formula = time ~ cleaner + type, data = poopdeck)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -59.743 -13.792  -0.683  13.583  83.583
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   54.357      1.705  31.881  < 2e-16 ***
## cleanerb      -0.420      2.088  -0.201 0.840665
## cleanerc      -6.940      2.088  -3.323 0.000944 ***
## typeshark     23.327      1.705  13.681  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 20.88 on 596 degrees of freedom
## Multiple R-squared:  0.2523,Adjusted R-squared:  0.2485
## F-statistic: 67.02 on 3 and 596 DF,  p-value: < 2.2e-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 54.357 minutes. Additionally, the average time to clean shark poop was 23.33 minutes slower than when cleaning parrrot poop.

[29] When you include an interaction term in a regression object, R will autmoatically include the main effects as well

```
##
## 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)          45.760      1.967  23.259  < 2e-16 ***
## cleanerb              8.060      2.782   2.897 0.003908 **
## cleanerc             10.370      2.782   3.727 0.000212 ***
## typeshark            40.520      2.782  14.563  < 2e-16 ***
## cleanerb:typeshark  -16.960      3.935  -4.310 1.91e-05 ***
## cleanerc:typeshark  -34.620      3.935  -8.798  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 19.67 on 594 degrees of freedom
## Multiple R-squared:  0.3385,Adjusted R-squared:  0.3329
## F-statistic: 60.79 on 5 and 594 DF,  p-value: < 2.2e-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.

Additional tips

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:

```
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   45.76 54.35667
## 2   1       b parrot   55   53.82 53.93667
## 3   1       c parrot   64   56.13 47.41667
## 4   1       a  shark  101   86.28 77.68333
## 5   1       b  shark   76   77.38 77.26333
## 6   1       c  shark   63   62.03 70.74333
```

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] 15.35173

# How far were the main effect model fits from the data on average?

mean(abs(poopdeck$me.fit - poopdeck$time))

## [1] 16.5351
```

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.

We can also see how far the models were on average from the data separately for each condition using dplyr.

```
library(dplyr)

poopdeck %>% group_by(cleaner, type) %>%
  summarise(
    int.fit.err = mean(abs(int.fit - time)),
    me.fit.err = mean(abs(me.fit - time))
  )

## Source: local data frame [6 x 4]
## Groups: cleaner [?]
##
##   cleaner   type int.fit.err me.fit.err
##     (chr)  (chr)       (dbl)      (dbl)
## 1       a parrot     14.2752   16.14407
## 2       a  shark     15.4944   17.36033
## 3       b parrot     13.9236   13.92127
## 4       b  shark     18.4552   18.45053
## 5       c parrot     15.4926   17.27167
## 6       c  shark     14.4694   16.06273
```

The results show that the interaction model had better fits (i.e.; lower errors) for virtually every condition

Repeated measures (linear mixed-effects) ANOVA using the lm4 packge

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[30].

```
# 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)
```

[30] For more tips on mixed-effects analyses, check out this great tutorial by Bodo Winter http://www.bodowinter.com/tutorial/bw_LME_tutorial2.

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 gropus 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()`:.

```
# 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 (see Figure 59), 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.

For more detail on the different types, check out https://mcfromnz.wordpress.com/2011/03/02/anova-type-iiiiii-ss-explained/

As you'll see, the `Anova()` function requres 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

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.
Figure 59: Testing if variables are balanced in an ANOVA.

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 appropirate 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 appopriate 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 indpendent 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.

13: 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 asssess 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.

Figure 60: Insert funny caption here.

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.

Linear regression with `lm()`

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`

lm()

`formula`

A formula in a 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.

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
9.3	0.88	4	182
11.1	1.05	5	191
8.7	0.85	6	176
10.4	1.15	5	195
10.6	0.92	5	182
12.3	0.44	4	183

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 ***
## 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 61 below:

You can access lots of different aspects of the regression object. To see what's inside, use `names()`

If you want to include all variables in a dataframe in the regression, you don't need to type the name of every independent variable in the `formula` argument, just enter a period (.) and R will assume you are using all variables.

```
# Shorten your formula with . to include
#  all variables:

diamonds.lm <- lm(formula = value ~ .,
                  data = diamonds)
```

Here are the main components of the summary of our regression object `diamonds.lm`"

Call — This just repeats our arguments to the `lm()` function

Residuals — Summary statistics of how far away the model fits are away from the true values. A residual is defined as the model fit minus the true value. For example, the median residual of -0.11 means that the median difference between the model fits and the true values is -0.11.

fficients — Estimates and inferential statistics on the beta values.

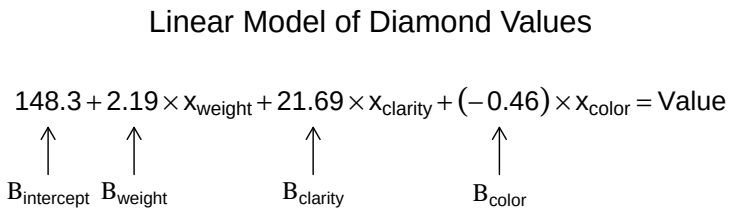


Figure 61: A linear model estimating the values of diamonds based on their weight, clarity, and color.

```
# 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 acecss 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
```

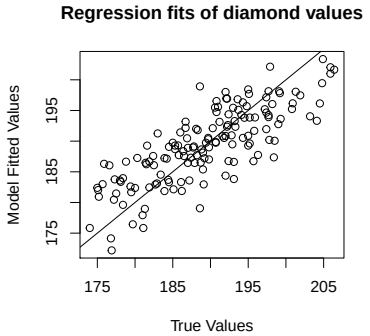
Getting model fits with `fitted.values`

To see the fitted values from a regresion 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:

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$value,
     y = diamonds.lm$fitted.values,
     xlab = "True Values",
     ylab = "Model Fitted Values",
     main = "Regression fits of diamond values"
     )

abline(b = 1, a = 0)
```



```
# 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
## 1   9.35    0.88     4 182.5    186.1
## 2  11.10    1.05     5 191.2    193.1
## 3   8.65    0.85     6 175.7    183.0
## 4  10.43    1.15     5 195.2    193.8
## 5  10.62    0.92     5 181.6    189.3
## 6  12.35    0.44     4 182.9    183.1
```

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.

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:

weight	clarity	color
20	1.5	5
10	0.2	2
15	5.0	3

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:

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 coeffiicent

```
# 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 200.53, 182.25, and 190.46 respectively according to our regression model.

*Including interactions in models: dv ~ 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)     157.472     10.569  14.899    0.000
## weight            0.979      1.070   0.915    0.362
## clarity           9.924     10.485   0.947    0.345
## weight:clarity    1.245      1.055   1.180    0.240
```

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 afterall? 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:

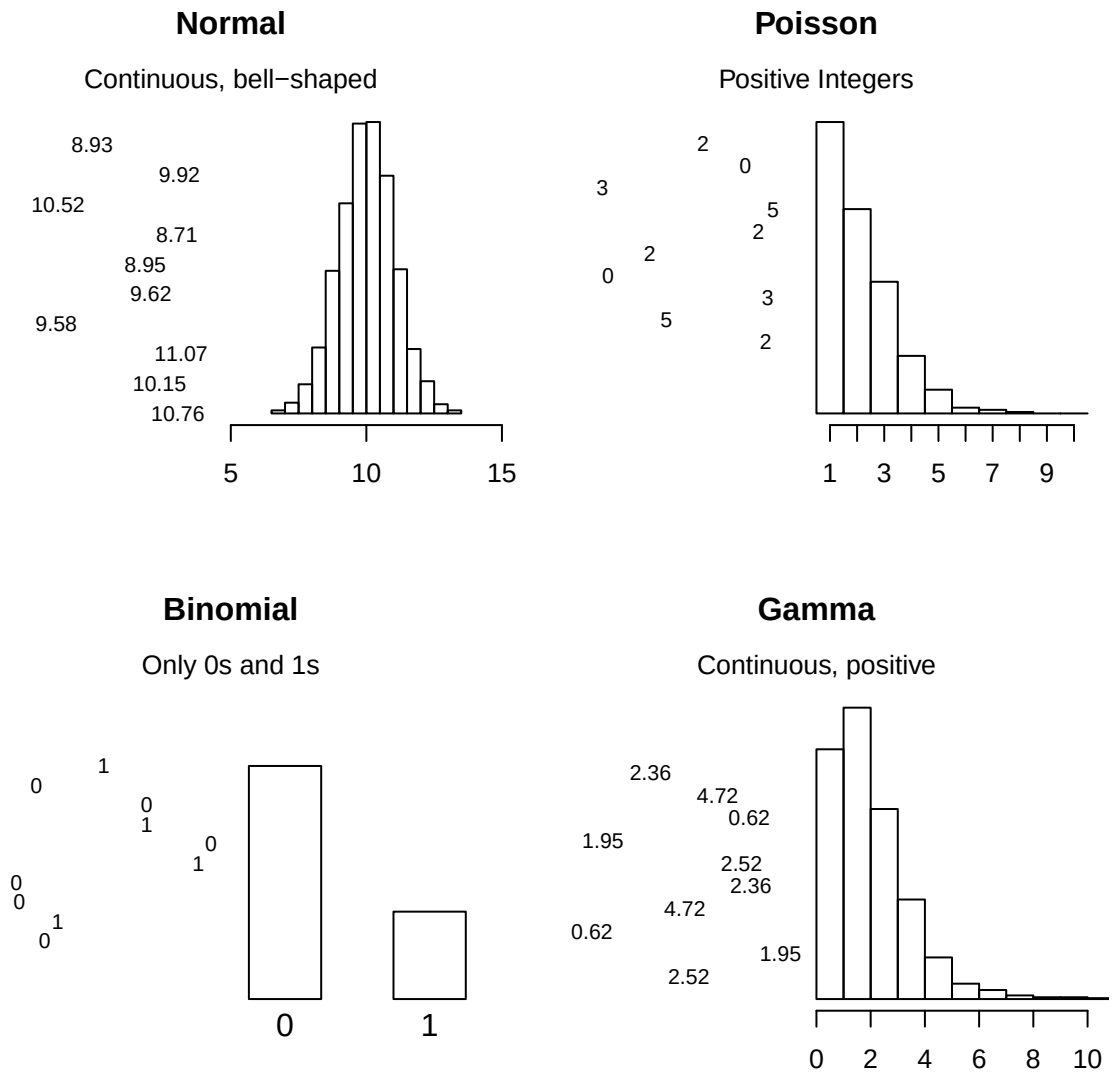
```
# Create centered versions of weight and clarity
diamonds$weight.c <- diamonds$weight - mean(diamonds$weight)
diamonds$clarity.c <- diamonds$clarity - mean(diamonds$clarity)

# 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      0.383  494.39     0.00
## weight.c              2.223      0.199   11.18     0.00
## clarity.c            22.248      2.134   10.43     0.00
## weight.c:clarity.c    1.245      1.055    1.18     0.24
```

Hey that looks much better! Now we see that the main effects are significant and the interaction is non-significant.

Regression on non-Normal data with `glm()`

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()`:



You can use the `glm()` function just like `lm()`. To specify the distribution of the dependent variable, use the `family` argument.

`glm()`

`function, data, subset`

The same arguments as in `lm()`

`family`

One of the following strings, indicating the link function for the general linear model

- "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 exponential response 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?"

```r
# Logit
logit.fun <- function(x) {1 / (1 + exp(-x))}

curve(logit.fun,
      from = -3,
      to = 3,
      lwd = 2,
      main = "Inverse Logit",
      ylab = "p(y = 1)",
      xlab = "Logit(x)"
      )

abline(h = .5, lty = 2)
abline(v = 0, lty = 1)
```

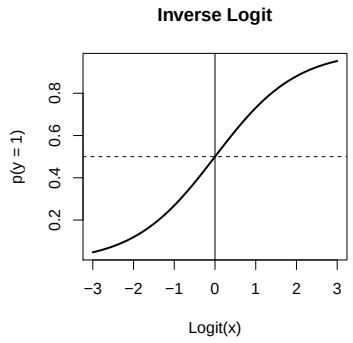


Figure 62: The inverse logit function used in binary logistic regression to convert logits to probabilities.

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.g175` indicating whether the value of a diamond is greater than 175 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.

```r
# 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)
```

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 the fitted values from the GLM back to the data
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 (see Figure **??**).

```
# 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.

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
```

Additional Tips

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 63) 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)
```

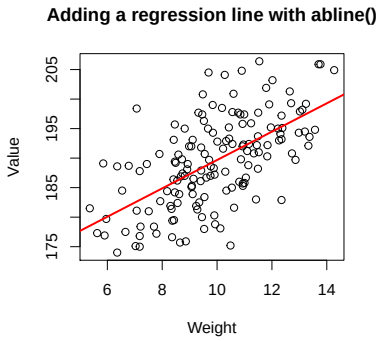


Figure 63: Adding a regression line to a scatterplot using `abline()`

Transforming skewed variables prior to standard regression

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 commmon transformation is a log-transformation.

For example, look at the distribution of movie revenues in the movies dataset in the margin Figure 64:

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.lt` defined as the logarithm of `revenue.all`

```
# Create a new log-transformed version of movie revenue
movies$revenue.all.lt <- log(movies$revenue.all)
```

In Figure 65 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 ok using this variable in a standard regression analysis with `lm()`.

```
# The distribution of movie revenus is highly
#  skewed.
hist(movies$revenue.all,
     main = "Movie revenue\nBefore log-transformatio
```

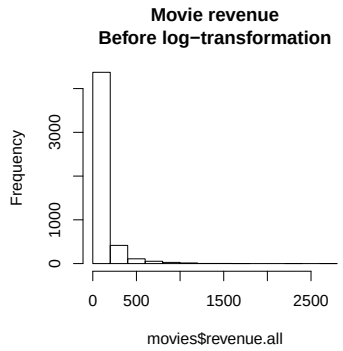


Figure 64: Distribution of movie revenues

```
# Distribution of log-transformed
#  revenue is much less skewed

hist(movies$revenue.all.lt,
     main = "Log-transformed Movie revenue")
```

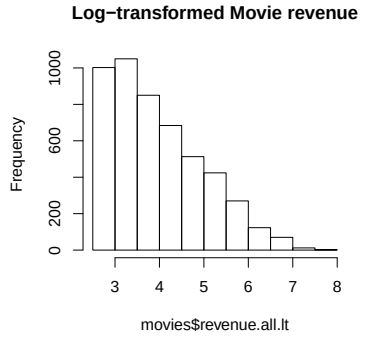


Figure 65: Distribution of movie revenues

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:

cannons	rooms	age	condition	color	style	jbb	price
18	20	140	5	red	classic	3976	3502
21	21	93	5	red	modern	3463	2955
20	18	48	2	plum	classic	3175	3281
24	20	81	5	salmon	classic	4463	4400
20	21	93	2	red	modern	2858	2177
21	19	60	6	red	classic	4420	3792

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?
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 displayed on the right in Figure 7
8. Using `price.all.blr`, predict the probability that the three new ships will have a selling price greater than 3500.

cannons	rooms	age	condition	color	style
12	34	43	7	black	classic
8	26	54	3	black	modern
32	65	100	5	red	modern

Figure 66: Data from 3 new ships about to be auctioned.

14: Writing your own functions

Figure 67: Functions. They're kind of a big deal.

Why would you want to write your own function?

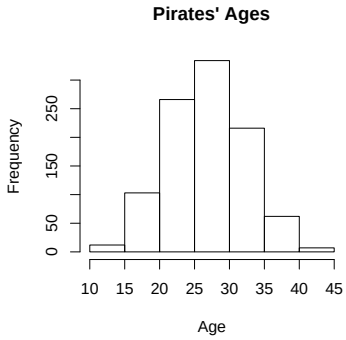
Throughout this book, you have been using tons of functions either built into base-R – like `mean()`, `hist()`, `t.test()`, or written by other people and saved in packages – like `pirateplot()` and `apa()`in the `yarrr` package. However, because R is a complete programming language, you can easily write 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. Check out the top figure on the right for an example. Instead of using these boring plots, you'd like to easily create a version like the bottom figure just below it. This plot not only has a more modern design, but it also includes statistical information, like the data mean, standard deviation, and a 95% confidence interval for the mean as a subheading. Now of course you know from chapter XX that you can customize plots in R any way that you'd like by adding customer parameter values like `col`, `bg` (etc.). But as you can see from the code above Figure , the plot has *exactly* the same inputs as the boring histogram just above it! No extra arguments necessary. How did I do that? Well as you may have guessed, I wrote a custom function called `my.hist()` that contains all the code necessary to build the plot. So now, anytime I want to make a fancy histogram, I can just use the `my.hist()` function rather than having to always write 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 once in R. If there's anything you like to do repeatedly in R, you will almost certainly like to write your own custom function to perform that action quickly and easily whenever you'd like. In fact, that's all functions really are, they're just chunks of R code that are stored behind the scenes for you to use without having to see (or write) the code yourself. 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.

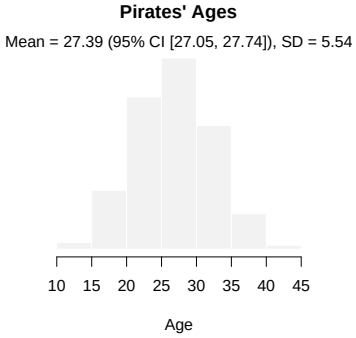
Boring standard histogram with default values of `hist()`:

```
hist(pirates$age,
     xlab = "Age",
     main = "Pirates' Ages")
```



Here's an advanced version made with our new function `my.hist()`!

```
my.hist(pirates$age,
        xlab = "Age",
        main = "Pirates' Ages")
```



The basic structure of a function

A function is simply an object that (usually) takes some input, 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 – like the ages of pirates – as an argument, and then returns the arithmetic mean of that vector as a single scaler 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. Inputs: 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 scaler statistic? A vector of data? A dataframe?

Here's how your function will look in R. You'll use two new functions, called `function()` and `return()`. As you can see, you put the function inputs as arguments to the `function()` function, and the output(s) as argument(s) to the `return()` function.

Yes, you use functions to create functions! Very Inception-y

```
NAME <- function(INPUTS) {

  ACTIONS

  return(OUTPUT)

  }
```

A simple example

Let's create a very simple example of a function. We'll create a function 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.

```
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] 3.75

mean(data)

## [1] 3.75
```

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...

If you ever want to see the exact code used to generate a function, you can just call the name of the function without the parentheses. For example, to see the code underlying our new function `my.mean` you can run the following:

```
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
##
## }
```

Using 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.

Including default values

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. 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.

You can use any kind of object as an input 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 input. 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 scalers.

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

```
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.

If you have a default value for every input, you can even call the function without specifying any inputs – R will set all of them to the default. For example, if we call `oh.god.how.much.did.i.spend` without specifying any inputs, R will set them all to 0 (which should make the result 0).

```
oh.god.how.much.did.i.spend()

## [1] 0
```

Using if/then statements in functions

Imagine a restaurant waiter who gives everyone crab juice no matter what they ordered. Or a guy who wears a tuxedo no matter if he's going to a dance or a barbeque. Well, unlike these guys, a good function needs to change what it does depending on what you give it. That is, before serving someone crab juice, a good `server()` function needs to first make sure that the order argument was actually crab juice (`order == "crabjuice"`).

You've already used functions that change based on their inputs. Consider the `summary()` function: If you evaluate the summary function on a numeric vector, it will return summary statistics of that vector. However, if you give the `summary` function a regression object, it will return output specific to regression objects – like estimates of the beta values. In other words, before actually calculating and displaying summary data, the `summary()` function first checks what type of input it received, and then calculates summary information specific to the type of input it is given.

Like all programming languages, R uses *if-then* statements to selectively run code based on some criteria. The function for running if-then statements is `if()` . The `if()` function has two main elements, a *logical test* and a chunk of *conditional code* (in curly braces). The chunk of code in the curly braces is conditional because it is *only* evaluated if the logical test is TRUE. If the logical test is `FALSE`, R will completely ignore the conditional code.

For example, in the following code chunk, I will define an object `a`

and assign the value TRUE to it. I will then write an if–statement that checks if a is TRUE, and if so, prints the statement `"a was true!"`:

```
a <- TRUE

if(a) { # If a is TRUE, do the following...

  print("a was true!")

} # End of conditional code

## [1] "a was true!"
```

Since a was TRUE, R executed the conditional code in the curly braces (and printed the statement). Now, If I run the code again, but set a to value of `FALSE`, the conditional code won't run, and the statement won't print:

```
a <- FALSE

if(a) {

  print("a was true!")

}
```

Of course, you don't have to explictly enter the value `TRUE` or `FALSE` as a logical test. You can put any R code in the logical test of an if statement as long as it returns a logical value of TRUE or FALSE.

Let's use `if()` statements in a really important function called `feed.me.negatives`. The function will have one input called x. The function will check if x is negative. If it is, the function will do one thing. If the input is not negative, it will do something else.

```
feed.me.negatives <- function(x) {

if(x < 0) {output <- "Yum! I love negative numbers!!"}

if(x > 0) {output <- "WTF WAS THAT?! I am feed.me.negatives, not be.a.douchebag"}

if(x == 0) {output <- "...I'm confused"}

return(output)

}
```

Let's test the function on some different inputs

```
feed.me.negatives(-2)

## [1] "Yum! I love negative numbers!!"

feed.me.negatives(10)

## [1] "WTF WAS THAT?! I am feed.me.negatives, not be.a.douchebag"

feed.me.negatives(0)

## [1] "...I'm confused"
```

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!

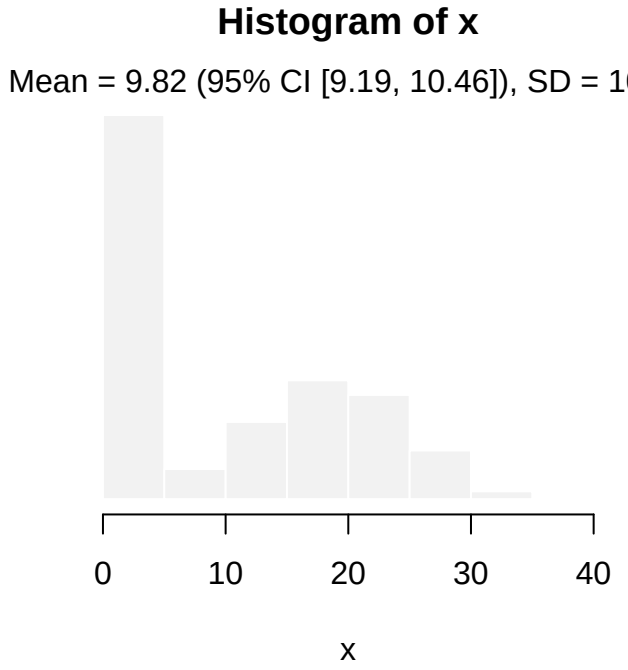
You'll notice that in the `show.me()` function I used the `my.hist()` function that we specified earlier in this chapter. If you try running `show.me(what = "plot")` without first defining `my.hist()`, the function will return an error!

```
show.me <- function(x, what) {

if(what == "plot") {

my.hist(x)
output <- "Ok! I hope you like the plot..."
}

if(what == "stats") {

output <- 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") {

output <- "I am a pirate, not your joke monkey."
}

return(output)
```

```
}
```

Let's test `show.me()` on the lengths of pirate beards. First, let's get it to create a histogram by setting `what = "plot"`

```
show.me(x = pirates$beard.length, what = "plot")
```




```
## [1] "Ok! I hope you like the 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 9.82 and the standard deviation be 10.24"
```

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.

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 "Custom_R_Functions.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:

```
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!

Tips and tricks for complex functions

Here are several tips and tricks that will help you to build good functions:

Conduct input quality checks

In most of your functions, you will expect the user to specify certain kinds of inputs. For example, in our `show.me` function, we only considered three possible inputs (`"plot"`, `"stats"`, `"tellmeajoke"`) for the `what` argument. If a user enters a different value for the `what` argument, the function won't do anything because none of the `if()` statements were satisfied. This can be frustrating for the user because he doesn't know if he did something wrong or if the function has a bug.

To help the user know which inputs are valid, you can include a quality check, where you explicitly check if the user's inputs are valid. If they valid, you can run the function as normal. If they are

not valid, you can return an error message to the user telling them what is wrong.

Let's add a quality control check to the `show.me` function. We'll do this by creating a logical value called `valid.input` which is TRUE when the input for `what` is valid, and FALSE when the input for `what` is not valid. Then, we'll define a warning message in the case where the input is not valid:

```
show.me <- function(x, what) {

valid.input <- what %in% c("plot", "stats", "tellmeajoke")

if(valid.input == TRUE) { # Begin TRUE input section

    if(what == "plot") {

    my.hist(x)
    output <- "Ok here's your histogram!"

    }

    if(what == "stats") {

    output <- (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") {

    output <- "I am a pirate, not your joke monkey."

    }

} # Close TRUE valid input section

if(valid.input == FALSE) { # Begin FALSE valid input section

    output <- "Bad what input. Please enter plot, stats, or tellmeajoke."

} # Close FALSE valid input section

return(output)
}
```

Let's try it out. We'll execute the `show.me()` function with an invalid value of `what`. Now, instead of being silent (like before), it should give us a warning telling us what went wrong:

```
show.me(x = 1, what = "surprise")

## [1] "Bad what input. Please enter plot, stats, or tellmeajoke."
```

Test your functions by hard-coding input values

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 == F]

  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!

Using ... as option inputs

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[31]. The ... input tells R that the user might add additional inputs that should be used later in the function.

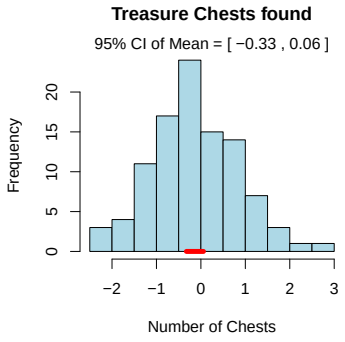
Here's a quick example, let's create a function called `hist.advanced` that plots a histogram with some optional additional arguments passed on with ...

[31] 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.

```
hist.advanced <- function(x, add.ci = T, ...) {

hist(x, # Main Data
     ... # Here is where the additional arguments go
     )

if(add.ci == T) {

  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()`. The result is in margin Figure . As you can see, R has passed our optional plotting arguments down to the main hist() function in the function code.

```
hist.advanced(x = rnorm(100), add.ci = T,
              main = "Treasure Chests found",
              xlab = "Number of Chests",
              col = "lightblue")
```



A worked example: Custom plotting functions

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

- add.mean: A logical value indicating whether or not to add vertical and horizontal lines at the mean value of x and y.

- add.regression: A logical value indicating whether or not to add a linear regression line
- sig.line.col: The color of the regression line if the slope is significant.
- nonsig.line.col: The color of the regression line if the slope is NOT significant.
- p.threshold: A numeric scaler indicating the p.value threshold for determining significance
- add.modeltext: A logical value indicating whether or not to include the regression equation as a sub-title to the plot

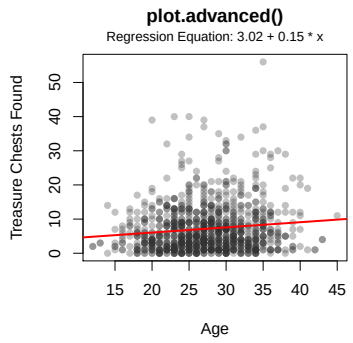
This plotting code is a bit long, but it's all stuff you've learned before.

```
plot.advanced <- function (x = rnorm(100),
                           y = rnorm(100),
                           add.mean = F,
                           add.regression = F,
                           sig.line.col = "red",
                           nonsig.line.col = "black",
                           p.threshold = .05,
                           add.modeltext = F,
                           ...  # 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 == T) {

  abline(h = mean(y), lty = 2)
  abline(v = mean(x), lty = 2)
}

# Add regression line if add.regression is TRUE
if(add.regression == T) {

  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 <- sig.line.col}
  if(p.value >= p.threshold) {line.col <- nonsig.line.col}

  abline(lm(y ~ x), col = line.col, lwd = 2) # Add regression line

}

  # Add regression equation text if add.modeltext is TRUE
if(add.modeltext == T) {
```

```
  # 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 the function on data from the pirates dataset. We'll put age on the x-axis and tchests on the y-axis. To see what the function can do, we'll turn on some of the additional elements, and include optional titles for the plot. You can see the result in the margin figure on the right!

```
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
              pch = 16,
              col = gray(.2, .3))
```



15: Loops

Figure 68: Loops in R can be fun. Just...you know...don't screw it up.

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 many pirates containing 100 yes/no questions. Question 1 might be "Do you ever shower?" and Question 2 might be "No seriously, do you ever shower?!" When you finish the survey, you could store the data as a dataframe with X rows (where X is the number of pirates you 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 "Y" or "N" 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...

```
survey.df$q.1[(survey.data$q1 %in% c("Y", "N")) == F] <- NA
survey.df$q.2[(survey.data$q2 %in% c("Y", "N")) == F] <- NA
# . ... Wait...I have to type this 98 more times?!
# .
# . ... My god this is boring...
# .
# .
# . ... I'm about to walk myself off the plank...
# .
survey.df$q.100[(survey.data$q100 %in% c("Y", "N")) == F] <- 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.

```
for(i in 1:100) { # Loop over all 100 columns

y <- survey.df[, i]  # Get data for ith column and save in a new object y

y[(y %in% c("Y", "N")) == F] <- NA # Convert invalid values in y to NA

survey.df[,column.i] <- y # Assign y back to survey.df!

} # 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.

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 an *index* that changes for every run of the loop. In R, the format of a for-loop is as follows:

```
for(INDEX.OBJECT in INDEX.VALUES) {

  LOOP.CODE

  }
```

As you can see, there are three key aspects of loops: The *index object*, the *index vector*, and the *loop code*:

1. **index object**: The object that is changing according to the index values. In the previous example, the index is just `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. **index vector**: A vector specifying all values that the index will take over the loop. In the previous example, the index values are `1:10`. You can specify the index values any way you'd like. 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 `c("Jolly Roger", "Black Pearl", "Queen Anne's Revenge")`.

3. **loop code**: The code that will be executed for all index values. In the previous example, the loop code is `print(i)`. You can write any R code you'd like in a loop - from plotting to analyses.

A simple loop: Adding integers from 1 to 100

Let's use a loop to add all the integers from 1 to 100. To do this, we'll start by creating an object called `current.sum` that contains the running sum of the numbers. This is where we'll store the running sum. We'll set the index object to i, the index 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:

```
current.sum <- 0

for(i in 1:100) {

 current.sum <- current.sum + i

}

current.sum

## [1] 5050
```

Looks like we get an answer of 5050.

Creating multiple plots with a loop

You can use loops to do much more than basic math. Oh no. One of the best uses of a loop is to put multiple graphs quickly and easily on the same chart. Let's use a loop to quickly create a matrix of four plots. For this example, we'll plot a histogram of the ages of pirates based on their favorite pixar movie from the pirates dataset.

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:

```
sum(1:100)

## [1] 5050
```

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:

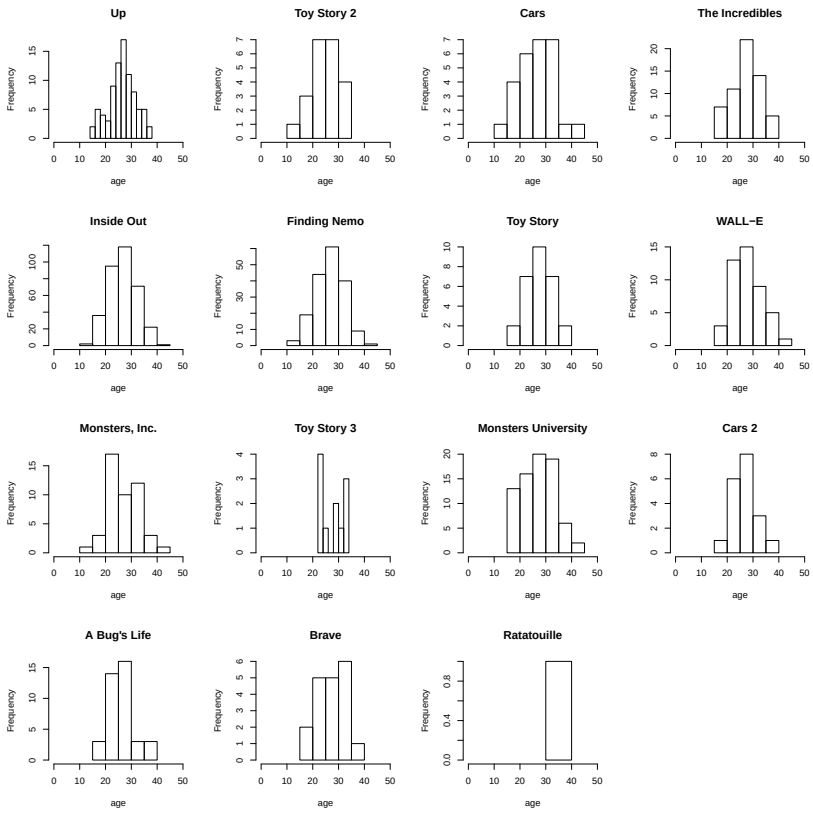
```
100 * 101 / 2

## [1] 5050
```

This boy grew up to be Gauss, a super legit mathematician.

Figure 69: Gauss. Guy's a total pirate. And totally would give us shit for using a loop to calculate the sum of 1 to 100...

```
par(mfrow = c(4, 4))  # Set up a 4 x 4 plotting space

# Create the index vector pixar.movies (all possible movies)
pixar.movies <- unique(pirates$fav.pixar)

for (movie.i in pixar.movies) {

  # subset data for current movie
  data.temp <- subset(pirates,
                      fav.pixar == movie.i)

  # Plot histogram of temporary data
  hist(data.temp$age,
       main = movie.i,
       xlab = "age",
       xlim = c(0, 50))
}
```

Here's how the plotting loop works. First, I set up a 4 x 4 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 defined the index vector as all the unique values of `fav.pixar` in the pirates dataframe. I assigned this object to `pixar.movies` which will later become our index vector. I then started the loop by setting the index object to `movie.i` and the index vector to `pixar.movies`. Next, I defined the loop code. In the loop code, I created a temporary data frame called `data.temp` which is a subset of the entire `pirates` data frame containing only rows for which the column `fav.pixar` is equal to the index object `movies.i`. Finally, I created a histogram of the age column from `data.temp`!



Storing sequential loop results in a container object

Figure 70: This is what I got when I googled "funny container".

One common use of loops is to create a table showing several statistics over some index. For example, in a survey of college students, you might want to calculate summary statistics for each level of sex, graduation year, major, etc. If you need to calculate a simple summary statistic (like a sample mean or median), you could easily do this using `aggregate()` or `dplyr`. However, for complex calculations, you may want to use a loop.

To store loop results, you'll need to start by setting up a container where the results will be stored. This container will usually be a vector, a data frame, or a list (we'll learn about lists later). Then, in your loop code, you will assign each result in your loop to an entry in this container.

Let's do a simple example. We'll create a vector called `squares`, where each entry in the vector is the square of the integers from one to ten.

The first thing we need to do is create a container vector called `squares` that is filled with NA values.

```
squares <- rep(NA, 10)
```

Now we can do our loop. Our index value will be i and our index vector will be 1:10. For our loop code, we'll calculate the square of the index value, and assign the result to the ith index of `squares`

```
for(i in 1:10) {

  result.i <- i ^ 2
  squares[i] <- result.i

}
```

Now let's look at the result. Hopefully we'll get the squares of 1 to 10...

```
squares
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Ok that loop was pretty simple but hopefully it shows you the basic idea of combining indexing with assignment in loops.

Again, there are easier ways to calculate the squares of integers than using a loop. Here, we can do the same thing using basic vector arithmetic:

```
squares <- (1:10) ^ 2
```

A complex data loop

Let's do a slightly more complex example. For this one, we'll use a loop to create a matrix containing many different statistics calculated

from the `pirates` dataset. Specifically, we'll create a matrix called `parrot.ci.df` that contains 95% confidence intervals for the mean number of parrots owned by pirates in each age group. That is, we want the 95% confidence interval of parrots owned by pirates who are 20, 21, ... 30. We can do this using a loop.

First, let's create the storage dataframe `parrot.ci.df`. To keep the dataframe small enough to print in this book, I'll restrict the age values to 20, 21, ... 30. However, you can include as many ages as you'd like:

```
# CREATING THE DATA CONTAINER parrot.ci.df

# Step 1 - Determine the age levels to analyze
ages.to.analyze <- 20:30
n.ages <- length(ages.to.analyze)

# Step 2 - Create the storage dataframe parrot.ci.df
parrot.ci.df <- data.frame(
  age = rep(NA, n.ages),
  mean = rep(NA, n.ages),
  lb = rep(NA, n.ages),
  ub = rep(NA, n.ages),
  stringsAsFactors = F
)
names(parrot.ci.df) <- c("age", "mean", "lb", "ub")
```

Here's how I did each step:

1. Step 1: Determine all the age values to analyze and assign them to the object `ages.to.analyze`. Then determine total number of unique age values to analyze and assign the total to the object `n.ages`.
2. Step 2: Create a storage dataframe called `parrot.ci.df` with N rows (where N is the number of ages we will analyze), and 3 columns (age, lower.bound, upper.bound). I started by filling `parrot.ci.df` with NA values. We will use the loop to replace these NA values with our statistics.

All done. Here's how the storage dataframe `parrot.ci.df` looks:

```
parrot.ci.df

##    age mean lb ub
## 1   NA   NA NA NA
## 2   NA   NA NA NA
## 3   NA   NA NA NA
## 4   NA   NA NA NA
## 5   NA   NA NA NA
## 6   NA   NA NA NA
## 7   NA   NA NA NA
## 8   NA   NA NA NA
## 9   NA   NA NA NA
## 10  NA   NA NA NA
## 11  NA   NA NA NA
```

Now that we have the storage dataframe, we can use a loop to replace the NA values with the values we want. For this loop, we'll loop over the rows of the dataframe `parrots.ci.df`.

```
# Step 3 - Define the index and index values
for (row.i in 1:nrow(parrot.ci.df)) {

# Step 4 - Calculate statistics for current index value
  age.i <- ages.to.analyze[row.i]
  data.temp <- subset(pirates, age == age.i) # Subset the original data
  mean.i <- mean(data.temp$parrots) # Get the sample mean
  ci.i <- t.test(data.temp$parrots) # Get the ci

# Step 5 - Assign statistics to parrot.ci.df
  parrot.ci.df$age[row.i] <- age.i
  parrot.ci.df$mean[row.i] <- mean.i
  parrot.ci.df$lb[row.i] <- ci.i[1]
  parrot.ci.df$ub[row.i] <- ci.i[2]

} # Close loop
```

Here's how I did each step:

1. Step 3: Set up the loop with an index of `row.i` where the index values are 1, 2, ... to the number of rows in the dataframe `result.df`. Our loop will analyze data specific to each row in the dataframe.
2. Step 4: As a function of the current index, determine the age value in that row, create a temporary dataset of data for that age value, then determine the sample mean and confidence interval for that temporary dataset.
3. Step 5: Assign the results for the current index value to `row.i` of the appropriate column in `result.df`

Let's look at the result to make sure it worked correctly. We should get a dataframe with 4 columns showing summary statistics for ages 20 through 30:

```
parrot.ci.df

##    age  mean    lb ub
## 1   20 1.667 5.493 35
## 2   21 2.265 6.265 33
## 3   22 2.150 6.954 39
## 4   23 2.300 7.893 59
## 5   24 1.797 7.154 63
## 6   25 2.324 10.41 67
## 7   26 2.415 6.463 52
## 8   27 2.702 8.656 83
## 9   28 2.400 8.053 59
## 10  29 2.532 7.218 61
## 11  30 2.947 7.409 74
```

Looks like it worked!

Loops over multiple indices

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.

Design Matrices

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 = F
                            )
```

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    31.69  2.437
## 2    CCCC   male         24    23.55  3.834
## 3   JSSFP female         34    33.67  3.151
## 4    CCCC female         27    26.07  3.521
```

Sweet! Our loop filled in the NA values with the statistics we wanted.

The list object

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....).

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.

```
number.list <- list("first" = rnorm(n = 10),
     "second" = rnorm(n = 5),
     "third" = rnorm(n = 15)
)

number.list

## $first
##  [1] -1.2430  0.1740 -1.1913 -0.5787 -1.2293  0.9176  1.6177 -2.0238
##  [9] -0.2475  0.9151
##
## $second
## [1] -1.62736 -1.10305 -0.52422  0.07205 -2.35107
##
## $third
```

```
##  [1] -0.65392  0.40617 -0.61255 -0.35520 -1.00425 -0.42765 -2.32361
##  [8] -0.69052 -0.42575 -1.04829  0.69998  0.74078 -0.08779  0.69702
## [15] -0.00156
```

To index 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]]`:

```
number.list[[1]]

##  [1] -1.2430  0.1740 -1.1913 -0.5787 -1.2293  0.9176  1.6177 -2.0238
##  [9] -0.2475  0.9151

number.list$second

## [1] -1.62736 -1.10305 -0.52422  0.07205 -2.35107
```

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:

```
samples.ls <- vector("list", 5)
```

If we look at `sample.ls`, we can see that it has 5 empty entries:

```
samples.ls

## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
```

Now, let's run the loop. For each run of the loop, we'll generate random samples and assign them to an object called `samples`. We'll then assign the `samples` object to the ith entry in `samples.ls`

```
for(i in 1:5) {
  samples <- rnorm(n = i, mean = 0, sd = 1)
  samples.ls[[i]] <- samples
}
```

Let's look at the result:

```
samples.ls

## [[1]]
## [1] 0.06156
##
## [[2]]
## [1] 0.9576 2.0565
##
## [[3]]
## [1]  0.48995  0.94618 -0.02269
##
## [[4]]
## [1]  1.4533  0.2751  0.6653 -0.9734
##
## [[5]]
## [1]  0.61423 -0.09755  0.77833 -0.20921  0.28763
```

Looks like it worked. The first element has one sample, the second element has two samples

If you want to convert a list to a vector format, you can use the command `unlist()`.

```
unlist(samples.ls)

##  [1]  0.06156  0.95758  2.05650  0.48995  0.94618 -0.02269  1.45330
##  [8]  0.27510  0.66534 -0.97345  0.61423 -0.09755  0.77833 -0.20921
## [15]  0.28763
```

Now all the results are compressed into one vector. Of course, the resulting vector has lost some information because you don't know which values came from which loop index.

When and when not to use loops

Loops are great because they save you a lot of code. However, a drawback of loops is that they can be slow relative to other functions.

For example, let's say we wanted to create a vector called `one.to.ten` that contains the integers from one to ten. We could do this using the following for-loop:

```
one.to.ten <- rep(NA, 10) # Create a dummy vector
for (i in 1:10) {one.to.ten[i] <- i} # Assign new values to vector
one.to.ten # Print the result

##  [1]  1  2  3  4  5  6  7  8  9 10
```

While this for-loop works just fine, you may have noticed that it's a bit silly. Why? Because R has built-in functions for quickly and easily calculating sequences of numbers. In fact, we used one of those functions in creating this loop! (See if you can spot it...it's `1:10`). The lesson is: before creating a loop, make sure there's not already a function in R that can do what you want.

Parallel computing with snowfall()

If you're running a long loop and find that it's taking a long time, you can try running in in parallel using `snowfall()`. Snowfall is a package that allows you to use multiple processors (called "slaves") on your computer to run a loop. Theoretically, this can increase the speed of your loop by 4, 16, 24 (etc.) times, but obviously it depends on how many cores your computer (or the server you're running the loop on) has.

To install the `snowfall` package, run the code

```
install.packages("snowfall")
```

There are 5 general steps to using `snowfall()`:

1. Load the `snowfall` package with `library("snowfall")`
2. Set up the slaves with `sfInit()`. Here, you determine how many slaves (processors) you want to run.
3. Send objects and libraries to the slaves with `sfExport()`, `sfExportAll()`, and/or `sfLibrary()`.
4. Write a function that each slave will evaluate (e.g.; `slave.fun`). The function must have a single input.
5. Run the slaves using `sfLapply(x, fun)`, or `sfSapply()`, where `x` are the values that the slaves will evaluate on the function `fun`. When the function is completed, it will return a list.
6. Turn the slaves off with `sfStop()`

Let's go through the five steps for a detailed example. In this example, we'll use two slaves to simultaneously calculate the average age of pirates who went to Jack Sparrow's School of Fashion and Piratry (JSSFP) and Captain Chunk's Cannon Crew (CCCC).

First, we'll load the `snowfall` package.

```
library("snowfall")

## Loading required package: snow
```

Second, we'll set up the cluster of slaves by running the `sfInit()` function. Enter the number of slaves you want to run using the `cpus` argument[32]. Once you execute this, R will prepare those slaves in the background.

[32] There isn't really a correct number of slaves to use. If you use too many, then each might run very slowly. Personally, I set the number of slaves to be 2 times the number of cores on my computer. My computer has 8 cores, so I usually use 16 slaves. But again, this might not be the optimal number. If you're concerned about speed, you can try using different numbers of slaves and see which number seems to do the job the fastest on your computer

```
sfInit(parallel = T, cpus = 2)

## R Version:  R version 3.2.4 (2016-03-10)

## snowfall 1.84-6.1 initialized (using snow 0.4-1): parallel
execution on 2 CPUs.
```

Third, you need to send objects and libraries to the slaves. These slaves are like brand new R sessions that are completely separate from your current R session. They won't have access to any of the objects you have defined or libraries you've loaded in your current session. To send objects and libraries to the slaves, use the sfExport and `sfExportAll` commands. If you run `sfExportAll()`, R will send *all* the objects in your current R session to the slaves. If you want to just send a few specific objects to the slaves, use `sfExport()`.

Let's send the `pirates` dataset to the slaves using `sfExport`. For some reason (that I don't quite understand), you need to name the objects as strings (with quotation marks). If you don't include them, the function won't work"

```
sfExport("pirates")

## Error in sfExport("pirates"): Unknown/unfound variable
pirates in export. (local=TRUE)
```

Now let's load a library in each of the slaves, use `sfLibrary()`. Let's load the `dplyr` package on the slaves (we actually don't need it for our loop, but I'll show you how to do it anyway):

```
sfLibrary(dplyr)

## Library dplyr loaded.
```

```
## Library dplyr loaded in cluster.
```

Fourth, we'll define the function that you want each slave to run. In this case, we'll create a function called `slave.fun()` that takes the name of a college as an input, and return the average age of pirates from that college as an output. Each slave will evaluate this function on a specific college.

```
slave.fun <- function(college.i) {

  data.temp <- subset(pirates, college == college.i)
  output <- mean(data.temp$age)

  return(output)

}
```

Fifth, now we're ready to run the cluster! We do this using the `sfSapply()`[33] function. There are two arguments to `sfSapply()`: x, a vector of index values sent to the function, and `fun`, a function that will be evaluated with each element of x. For this example, we'll set x to a vector of the two colleges, and the function to be `slave.fun`. The function will then send each value of the vector x to a slave, the slave will evaluate `slave.fun()` for a value of x, and return the result to the main R session:

[33] In addition to `sfSapply()`, you can also use `sfLapply()`, which returns stores the results as a list

```
cluster.result <- sfSapply(x = c("JSSFP", "CCCC"), fun = slave.fun)

## Error in checkForRemoteErrors(val): 2 nodes produced errors;
first error: object 'pirates' not found
```

Sixth, now that the cluster is finished, we need to close the clusters with the command `sfStop()`.

```
sfStop()

##
  ## Stopping cluster
```

Now that we're finished, we can look at the result. It should be a vector of length two, where the first element is the mean age of pirates who went to JSSFP and the second element is the mean age of pirates who went to CCCC:

```
cluster.result

## Error in eval(expr, envir, enclos): object 'cluster.result'
not found
```

Looks like our result is what we want. Now, of course this was a very simple example, and it would have been much easier to use the `aggregate()` function to accomplish this task. But the general structure should allow you to perform much more complicated loops in parallel.

Additional tips for using Snowfall

1. **Run your simulations in the background!** While you're using snowfall (or running any processor intensive code), you won't be able to continue executing any code in R. However, if you need to scratch your R itch during a long simulation, you can open up a second instance of R that will run independently of any other R sessions. On a Mac, you can do this by opening up the Terminal (in the Applications folder) and running the command `open -n -a RStudio.app`. When you run this command, another instance of RStudio should open up that you can now use while your simulation is running

2. **Print a simulation progress report**. If your snowfall loop is taking a long time to run, you can get a progress report that tells you what percentage of the simulation is completed. While this won't speed anything up, it will give you an idea of how fast things are moving and help you know when it might be finished. To do this, use the `perUpdate` argument, which takes an integer as an argument, and prints a notification for every ith% run of the simulation. The `perUpdate` argument only works in the the cluster function `sfClusterApplySR()` and not in `sfSapply()`, but the functions are virtually identical. For example, let's say we want to execute a slave function 1,000,000 times. We can get a progress report for every 1% of simulations (every 10,000 instances) by using the following code:

```
cluster.result <- sfClusterApplySR(1:1000000, # Execute the function from 1 to 1,000,000
                                   fun = slave.fun, # The slave function
                                   perUpdate = 1 # Print an update for every 1% completion of the simulati
                                   )
```

16: Data Cleaning and preparation

In this chapter, we'll cover many tips and tricks for preparing a dataset for analysis - from recoding values in a dataframe, to merging two dataframes together, to ... lots of other fun things. Some of the material here has been covered in other chapters - however, because data preparation is so important and because so many of these procedures go together, I thought it would be wise to have them all in one place.

The Basics

Changing column names in a dataframe

To change the names of the columns in a dataframe, use `names()`, indexing, and assignment. For example, to change the name of the first column in a dataframe called `data` to `"participant"`, you'd do the following:

```
names(data)[1] <- "participant"
```

If you don't know the exact index of a column whose name you want to change, but you know the original name, you can use logical indexing. For example, to change the name of a column titled `sex` to `gender`, you can do the following:

```
names(data)[names(data) == "sex"] <- "gender"
```

Changing the order of columns

You can change the order of columns in a dataframe with indexing and reassignment. For example, consider the ChickWeight dataframe which has four columns in the order: weight, Time, Chick and Diet

```
ChickWeight[1:2,]
```

```
## Grouped Data: weight ~ Time | Chick
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
```

As you can see, weight is in the first column, Time is in the second, Chick is in the third, and Diet is in the fourth. To change the column order to, say: Chick, Diet, Time, weight, we'll create a new column index vector called `new.order`. This vector will put the original columns in their new places. So, to put the Chick column first, we'll start with a column index of 3 (the original Chick column). To put the Diet column second, we'll put the column index 4 next (etc.):

```
new.order <- c(3, 4, 2, 1)
# Chick, Diet, Time, weight
```

Now, we'll use reassignment to put the datframe in the new order:

```
ChickWeight <- ChickWeight[,new.order]
```

Here is the result:

```
ChickWeight[1:2,]

## Grouped Data: weight ~ Time | Chick
##   Time Chick weight Diet
## 1    0     1     42    1
## 2    2     1     51    1
```

Instead of using numerical indices, you can also reorder the columns using a vector of column names. Here's another index vector containing the names of the columns in a new order:

```
new.order <- c("Time", "weight", "Diet", "Chick")
```

Here is the result:

```
ChickWeight <- ChickWeight[,new.order]
ChickWeight[1:2,]

## Grouped Data: weight ~ Time | Chick
##   Time weight Diet Chick
## 1    0     42    1     1
## 2    2     51    1     1
```

If you only want to move some columns without changing the others, you can use the following trick using the `setdiff()` function. For example, let's say we want to move the two columns Diet and weight to the front of the `ChickWeight` dataset. We'll create a new index by combining the names of the first two columns, with the names of all remaining columns:

```
# Define first two column names
to.front <- c("Diet", "weight")

# Get rest of names with setdiff()
others <- setdiff(names(ChickWeight),
                  to.front)

# Index ChickWeight with c(to.front, others)
ChickWeight <- ChickWeight[c(to.front, others)]

# Check Result
ChickWeight[1:2,]

## Grouped Data: weight ~ Time | Chick
##   Diet weight Time Chick
## 1    1     42    0     1
## 2    1     51    2     1
```

If you are working with a dataset with many columns, this trick can save you a lot of time

Converting values in a vector or dataframe

In one of the early chapters of this book, we learned how to change values in a vector item-by-item. For example, to change all values of 1 in a vector called vec to 0, we'd use the code:

```
vec[vec == 1] <- 0
```

However, if you need to convert many values in a vector, typing this type of code over and over can become tedious. Instead, I recommend writing a function to do this. As far as I know, this function is not stored in base R, so we'll write one ourselves called `recodev()` (which stands for 'recode vector'). The `recodev` function is included in the `yarrr` package. If you don't have the package, the full definition of the function is on the sidebar. – just execute the code in your R session to use it. The function has 4 inputs

- `original.vector`: The original vector that you want to recode
- `old.values`: A vector of values that you want to replace. For example, `old.values = c(1, 2)` means that you want to replace all values of 1 or 2 in the original vector.
- `new.values`: A vector of replacement values. This should be the same length as `old.values`. For example, `new.values = c("male", "female")` means that you want to replace the two values in `old.values` with "male" and "female".
- `others`: An optional value that is used to replace any values in `original.vector` that are not found in `old.values`. For example, `others = NA` will convert any values in `original.vector` not found in `old.values` to NA. If you want to leave other values in their original state, just leave the `others` argument blank.

Here's the function in action: Let's say we have a vector gender which contains 0s, 1s, and 2s

```
gender <- c(0, 1, 0, 1, 1, 0, 0, 2, 1)
```

Let's use the `recodev()` function to convert the 0s to "female", 1s to "male" and 2s to "other"

```
gender <- recodev(original.vector = gender,
         old.values = c(0, 1, 2),
         new.values = c("female", "male", "other")
)
```

Now let's look at the new version of gender. The former values of 0 should now be female, 1 should now be male, and 2 should now be other:

```
#recodev function
# Execute this code in R to use it!

recodev <- function(original.vector,
                    old.values,
                    new.values,
                    others = NULL) {

if(is.null(others)) {

  new.vector <- original.vector

}

if(is.null(others) == F) {

 new.vector <- rep(others,
                   length(original.vector))

}

for (i in 1:length(old.values)) {

change.log <- original.vector == old.values[i] &
              is.na(original.vector) == F

new.vector[change.log] <- new.values[i]

}

return(new.vector)

}
```

```
gender

## [1] "female" "male"   "female" "male"   "male"   "female" "female" "other"
## [9] "male"
```

Changing the class of a vector

If you would like to convert the class of a vector, use a combination of the `as.numeric()` and `as.character()` functions.

For example, the following dataframe called `data`, has two columns: one for age and one for gender.

```
data <- data.frame("age" = c("12", "20", "18", "46"),
                   "gender" = factor(c("m", "m", "f", "m")),
                   stringsAsFactors = F
                   )
str(data)

## 'data.frame': 4 obs. of  2 variables:
##  $ age   : chr  "12" "20" "18" "46"
##  $ gender: Factor w/ 2 levels "f","m": 2 2 1 2
```

As you can see, age is coded as a character (not numeric), and gender is coded as a factor. We'd like to convert age to numeric, and gender to character.

To make age numeric, just use the `as.numeric()` function:

```
data$age <- as.numeric(data$age)
```

To convert `gender` from a factor to a character, use the `as.character()` function:

```
data$gender <- as.character(data$gender)
```

Let's make sure it worked:

```
str(data)

## 'data.frame': 4 obs. of  2 variables:
##  $ age   : num  12 20 18 46
##  $ gender: chr  "m" "m" "f" "m"
```

Now that age is numeric, we can apply numeric functions like `mean()` to the column:

If we try to calculate the `mean()` of age without first converting it to numeric, we'll get an error:

```
# We'll get an error because age is not
# numeric (yet)
mean(data$age)

## Warning in
mean.default(data$age): argument
is not numeric or logical: returning
NA

## [1] NA
```

```
mean(data$age)

## [1] 24
```

Splitting numerical data into groups using cut()

When we create some plots and analyses, we may want to group numerical data into bins of similar values. For example, in our pirate survey, we might want to group pirates into age decades, where all pirates in their 20s are in one group, all those in their 30s go into another group, etc. Once we have these bins, we can calculate aggregate statistics for each group.

R has a handy function for grouping numerical data called `cut()`

`cut()`

`x`

A vector of numeric data

`breaks`

Either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which x is to be cut. For example, `breaks = 1:10` will put break points at all integers from 1 to 10, while `breaks = 5` will split the data into 5 equal sections.

`labels`

An optional string vector of labels for each grouping. By default, labels are constructed using "(a,b]" interval notation. If labels = FALSE, simple integer codes are returned instead of a factor.

`right`

A logical value indicating if the intervals should be closed on the right (and open on the left) or vice versa.

Let's try a simple example by converting the integers from 1 to 50 into bins of size 10:

```
cut(1:50, seq(0, 50, 10))

##  [1] (0,10]  (0,10]  (0,10]  (0,10]  (0,10]  (0,10]  (0,10]  (0,10]
##  [9] (0,10]  (0,10]  (10,20] (10,20] (10,20] (10,20] (10,20] (10,20]
## [17] (10,20] (10,20] (10,20] (10,20] (20,30] (20,30] (20,30] (20,30]
```

```
## [25] (20,30] (20,30] (20,30] (20,30] (20,30] (20,30] (30,40] (30,40]
## [33] (30,40] (30,40] (30,40] (30,40] (30,40] (30,40] (30,40] (30,40]
## [41] (40,50] (40,50] (40,50] (40,50] (40,50] (40,50] (40,50] (40,50]
## [49] (40,50] (40,50]
## Levels: (0,10] (10,20] (20,30] (30,40] (40,50]
```

As you can see, our result is a vector of factors, where the first ten elements are (0, 10], the next ten elements are (10, 20], and so on. In other words, the new vector treats all numbers from 1 to 10 as being the same, and all numbers from 11 to 20 as being the same.

Let's test the `cut()` function on the age data from `pirates`. We'll add a new column to the dataset called `age.decade`, which separates the age data into bins of size 10. This means that every pirate between the ages of 10 and 20 will be in the first bin, those between the ages of 21 and 30 will be in the second bin, and so on. To do this, we'll enter `pirates$age` as the x argument, and `seq(10, 60, 10)` as the `breaks` argument:

```
pirates$age.decade <- cut(
  x = pirates$age,  # The raw data
  breaks = seq(10, 60, 10)  # The break points of the cuts
  )
```

To show you how this worked, let's look at the first few rows of the columns `age` and `age.cut`

```
head(pirates[c("age", "age.decade")])

##   age age.decade
## 1  30    (20,30]
## 2  25    (20,30]
## 3  25    (20,30]
## 4  29    (20,30]
## 5  31    (30,40]
## 6  30    (20,30]
```

As you can see, `age.cut` has correctly converted the original age variable to a factor.

From these data, we can now easily calculate how many pirates are in each age group using `table()`

```
table(pirates$age.decade)

##
##   (0,10]  (10,20]  (20,30]  (30,40]  (40,50]  (50,60]  (60,70]  (70,80]
##        0      115      600      278        7        0        0        0
##  (80,90] (90,100]
##        0        0
```

Once you've used `cut()` to convert a numeric variable into bins, you can then use `aggregate()` or dplyr to calculate aggregate statistics for each bin. For example, to calculate the mean number of tattoos of pirates in their 20s, 30s, 40s, ... we could do the following:

```
# Calculate the decade for each pirate

pirates$age.decade <- cut(
  pirates$age,
  breaks = seq(0, 100, 10)
  )

# Calculate the mean number of tattoos
#  in each decade

aggregate(tattoos ~ age.decade,
          FUN = mean,
          data = pirates)

##   age.decade tattoos
## 1    (10,20]   9.722
## 2    (20,30]   9.320
## 3    (30,40]   9.335
## 4    (40,50]   8.143
```

Merging two dataframes

Merging two dataframes together allows you to combine information from both dataframes into one. For example, a teacher might have a dataframe called `students` containing information about her class. She then might have another dataframe called `exam1scores` showing the scores each student received on an exam. To combine these data into one dataframe, you can use the `merge()` function. For those of you who are used to working with Excel, `merge()` works a lot like `vlookup` in Excel:

`merge()`

`x, y`

2 dataframes to be merged

`by, by.x, by.y`

The names of the columns that will be used for merging. If the the merging columns have the same names in both dataframes, you can just use `by = c("col.1", "col.2"...)`. If the merging columns have different names in both dataframes, use `by.x` to name the columns in the x dataframe, and `by.y` to name the columns in the y dataframe. For example, if the merging column is called `STUDENT.NAME` in dataframe x, and `name` in dataframe y, you can enter `by.x = "STUDENT.NAME", by.y = "name"`

`all.x, all.y`

A logical value indicating whether or not to include non-matching rows of the dataframes in the final output. The default value is `all.y = FALSE`, such that any non-matching rows in y are not included in the final merged dataframe.

A generic use of `merge()`, looks like this:

```
new.df <- merge(x = df.1,  # First dataframe
                y = df.2,  # Second dataframe
                by = "column"  # Common column name in both x and y
                )
```

where `df.1` is the first dataframe, `df.2` is the second dataframe, and `"column"` is the name of the column that is common to both dataframes.

For example, let's say that we have some survey data in a dataframe called `survey`.

Here's how the survey data looks:

```
survey

##    pirate   country
## 1       1   Germany
## 2       2  Portugal
## 3       3     Spain
## 4       4   Austria
## 5       5 Australia
## 6       6   Austria
## 7       7   Germany
## 8       8  Portugal
## 9       9  Portugal
## 10     10   Germany
```

```
survey <- data.frame(
  "pirate" = 1:10,
  "country" = c("Germany",
                "Portugal",
                "Spain",
                "Austria",
                "Australia",
                "Austria",
                "Germany",
                "Portugal",
                "Portugal",
                "Germany"
                ),
  stringsAsFactors = F
)
```

Now, let's say we want to add some country-specific data to the dataframe. For example, based on each pirate's country, we could add a column called `language` with the pirate's native language, and `continent` – the continent the pirate is from. To do this, we can start by creating a new dataframe called `country.info`, which tell us the language and continent for each country:

Let's take a look at the `country.info` dataframe:

```
country.info

##     country  language continent
## 1   Germany    German    Europe
## 2  Portugal Portugese    Europe
## 3     Spain   Spanish    Europe
## 4   Austria    German    Europe
## 5 Australia   English Australia
```

```
country.info <- data.frame(
"country" = c("Germany", "Portugal",
              "Spain", "Austria", "Australia"),
"language" = c("German", "Portugese",
               "Spanish", "German", "English"),
"continent" = c("Europe", "Europe", "Europe",
                "Europe", "Australia"),
stringsAsFactors = F)
```

Now, using `merge()`, we can combine the information from the `country.info` dataframe to the `survey` dataframe:

```
survey <- merge(survey,
                country.info,
                by = "country"
                )
```

Let's look at the result!

```
survey

##      country pirate  language continent
## 1  Australia      5   English Australia
```

```
## 2    Austria      4    German    Europe
## 3    Austria      6    German    Europe
## 4    Germany      1    German    Europe
## 5    Germany     10    German    Europe
## 6    Germany      7    German    Europe
## 7   Portugal      8 Portugese    Europe
## 8   Portugal      9 Portugese    Europe
## 9   Portugal      2 Portugese    Europe
## 10     Spain      3   Spanish    Europe
```

As you can see, the `merge()` function added all the `country.info` data to the `survey` data.

Random Data Preparation Tips

Appendix

```
plot(1, xlim = c(0, 26), ylim = c(0, 26),
     type = "n", main = "Named Colors", xlab = "", ylab = "",
     xaxt = "n", yaxt = "n")

rect(xleft = rep(1:26, each = 26)[1:length(colors())] - .5,
     ybottom = rep(26:1, times = 26)[1:length(colors())] - .5,
     xright = rep(1:26, each = 26)[1:length(colors())] + .5,
     ytop = rep(26:1, times = 26)[1:length(colors())] + .5,
     col = colors()
     )

text(x = rep(1:26, each = 26)[1:length(colors())],
     y = rep(26:1, times = 26)[1:length(colors())],
     labels = colors(), cex = .3
     )
```

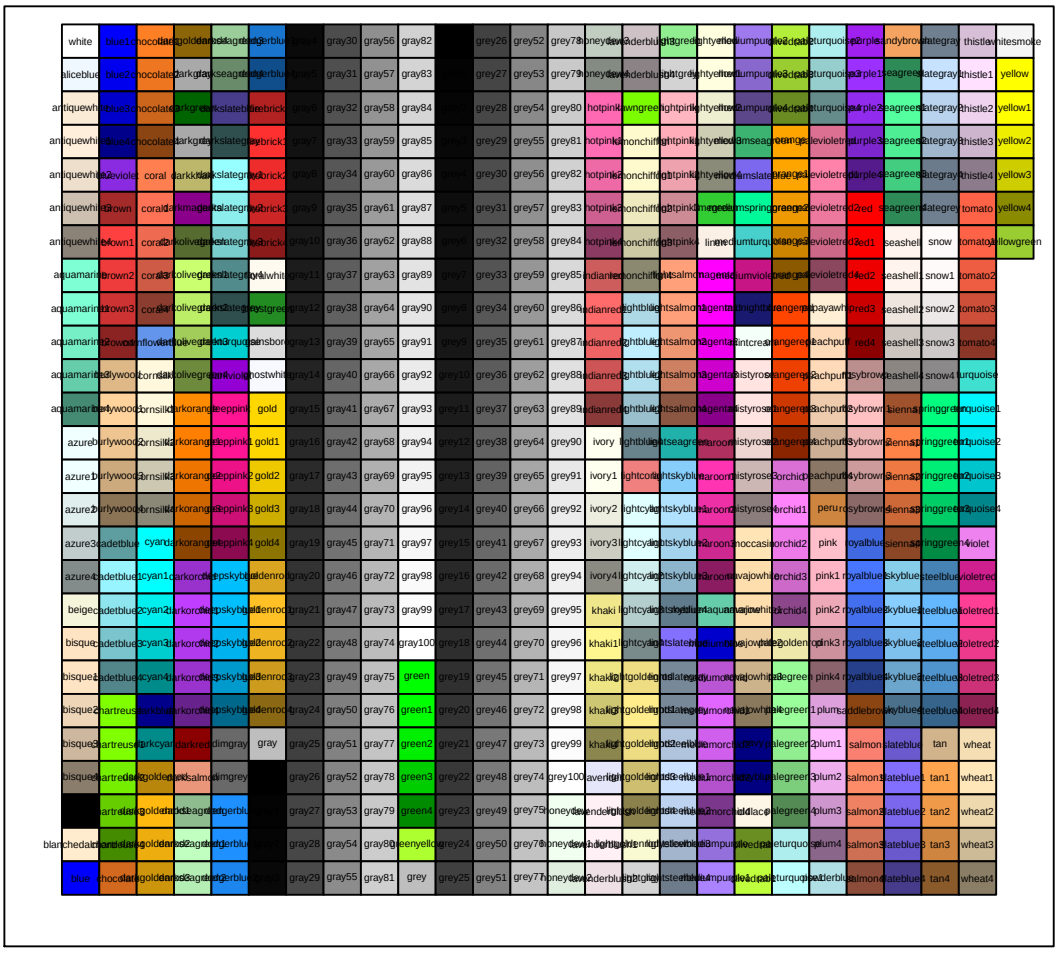


Figure 71: The colors stored in `colors()`.

Index