YaRrr!

The Pirate's Guide to R

DR. NATHANIEL D. PHILLIPS

YARRR! THE PIRATE'S GUIDE TO R

Copyright © 2016 Dr. Nathaniel D. Phillips

PUBLISHED BY

http://www.thepiratesguidetor.com

This document may not be used for any commercial purposes. All rights are reserved by Nathaniel Phillips.

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, and 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.

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

3. 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 wrote 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.

4. 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[2]. 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!

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

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

```
require("circlize")


## Loading required package: circlize
## Warning: package 'circlize' was built under R version
3.2.3

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

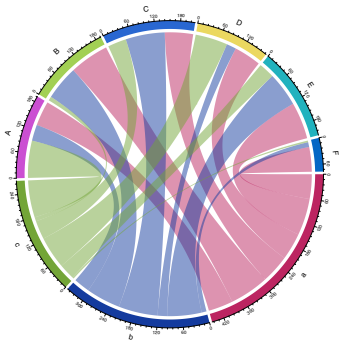


Figure 1: This is a `chordDiagram` plot that comes with the R package `circlize`. How fucking cool is this?! Have fun trying to make this with SPSS.

[2] Get used to the bad jokes people. Lots more where that came from.

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.

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.

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

R is like a relationship...

Figure 2: Yep, R will become both your best friend and your worst nightmare. The bad times will make the good times oh so much sweeter.

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.

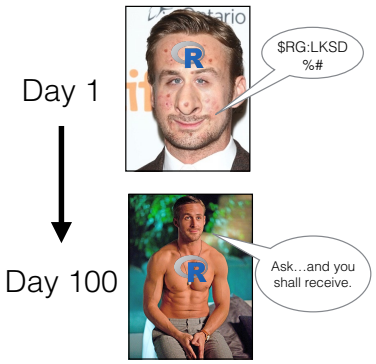


Figure 3: When you first meet R, it will look so fugly that you'll wonder if this is all some kind of sick joke. But trust me, once you learn how to talk to it, and clean it up a bit, all your friends will be crazy jealous.

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 4 (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.

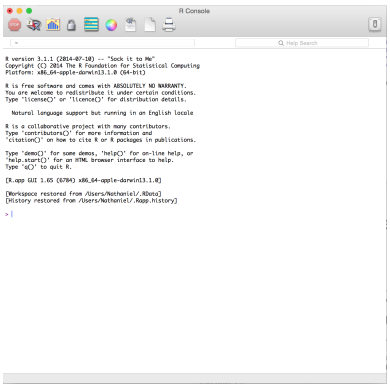


Figure 4: Here is how the standard R application looks. Not too exciting - just how we like it!

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!

The four RStudio windows

When you open RStudio, you'll see the following four windows (also called panes) shown in in Figure 5. However, your windows might be in a different order that those in Figure 5. If you'd like, you can change the order of the windows under RStudio preferences. You can also change their shape by either clicking the minimize or maximize buttons on the top right of each panel, or by clicking and dragging the middle of the borders of the windows.

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

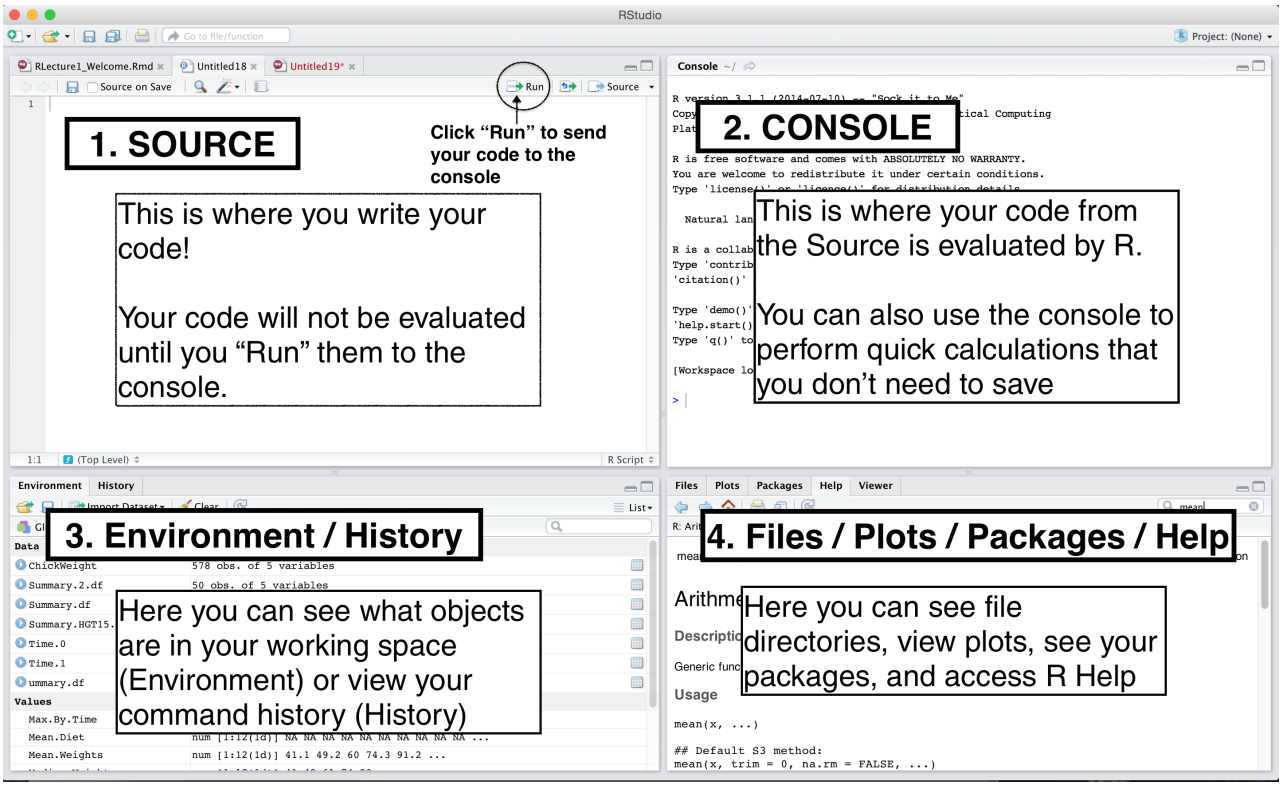


Figure 5: The four panes of RStudio.

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% 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.

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.

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:

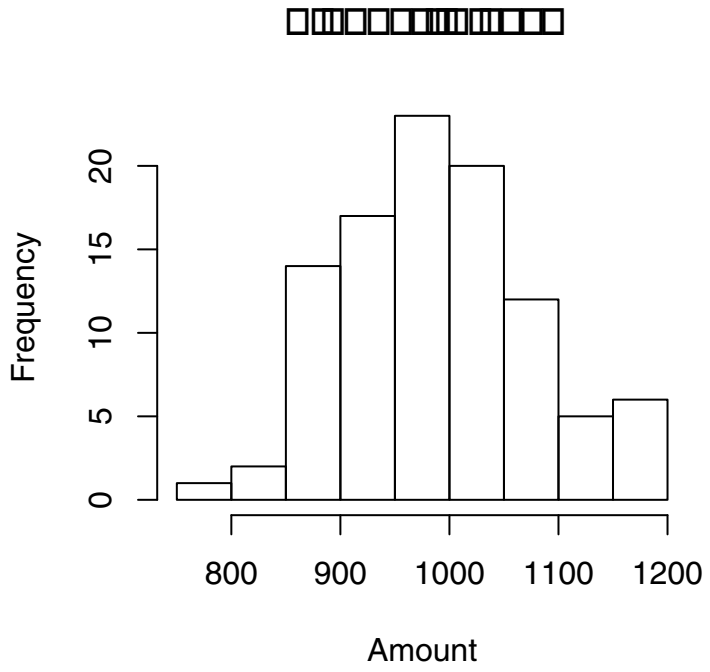
1. Files - The files panel gives you access to the file directory on your harddrive. One nice feature of the "Files" panel is that you can use it to set your working directory - once you navigate to a folder you want to read and save files to, click "More" and then "Set As Working Directory." We'll talk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

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.

```
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:

```
?hist
```

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

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.

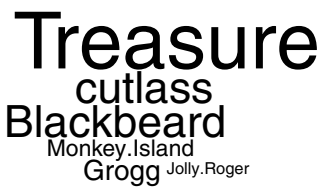
```
install.packages("wordcloud")
```

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[3] 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:

[3] 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
```

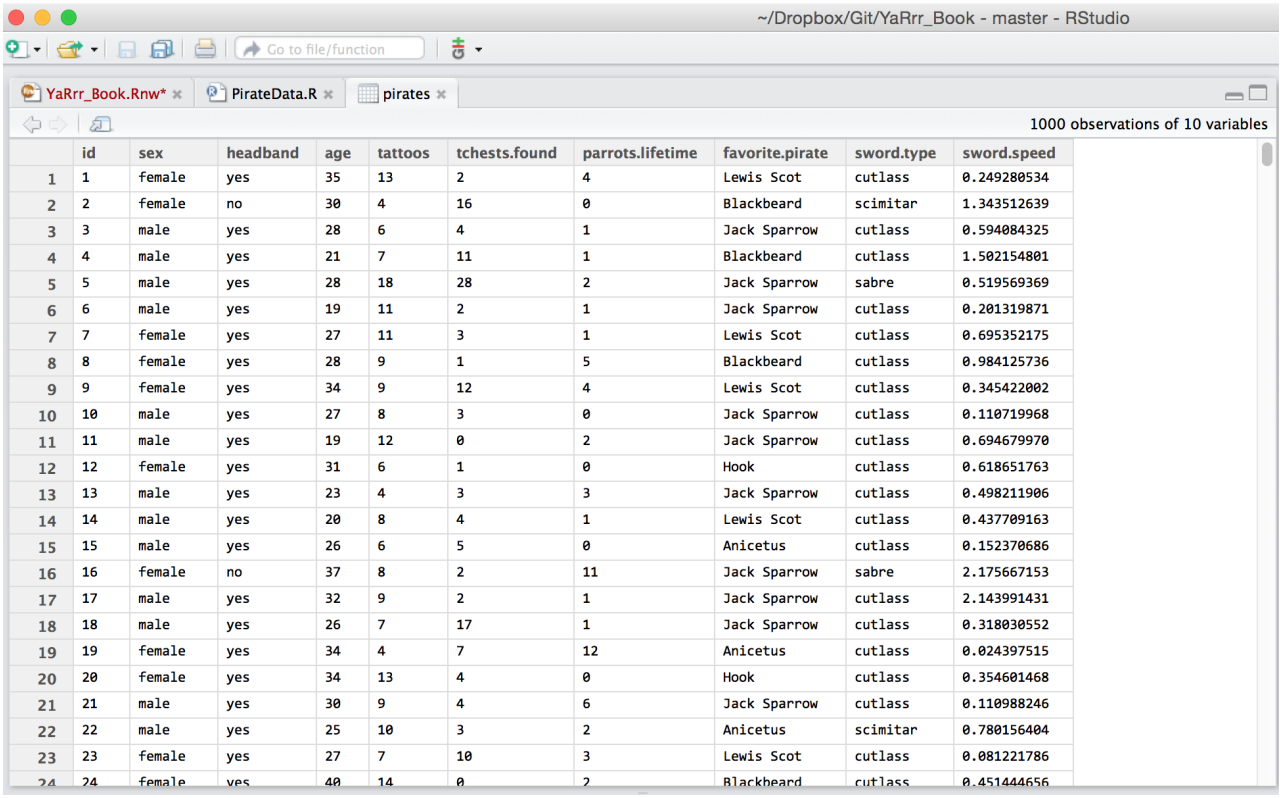
If everything went correctly, you should have access to all the datasets mentioned in this book in addition to many functions like `apa()` and `pirateplot()`.

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 6: The pirates dataset included in the yarrr package.

If you were unable to download the yarrr package and access the pirates dataset (this can happen on some computers depending on many possible issues), you can also use the following code to download the dataset directly from the web:

```
pirates <- read.table(file = "http://nathanieldphillips.com/wp-content/uploads/2015/11/pirates1.txt",
                      header = T,
                      sep = "\t", # tab-delimited
                      stringsAsFactors = F
                      )
```

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.

2: Coding Basics

Chapter Goals

1. Accept that learning R will take time (and promise you'll never go back to SPSS!)
2. Know how to use comments and spaces in R code.
3. Be able to define and manipulate scalars and vectors
4. Generate vectors using c(), :, rep(), and seq()

The basics of R programming

Ok, let's write some code! Again, we will write all our code in a script file in the Source pane of RStudio. When we want to execute it, we'll send it to the Console.

Figure 7: 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.

Tip: To execute code from the source to the console, highlight it and use the hot-keys "Command-Return" on Mac or "Control-Enter" on PC.

R as a calculator

At its heart, R is just a fancy calculator. Let's do some basic algebra, type the following command into the source, then highlight the text and click "Run" to execute it in the console:

```
1+1

## [1] 2
```

As you can see, R returns the (thankfully correct) value of 2. You'll notice that the console also returns the text [1]. This is just telling you you the index of the value next to it. Don't worry about this for now, it will make more sense later.

Commenting code with the (pound) sign

Additionally, you'll notice that I included a comment in the code using the # sign. R will ignore everything on a line after the # sign. So why do we use comments? Mainly to explain to others, including your future self, what you are trying to do with your code.

Do your future self a favor and use comments to explain what you're doing with your code. Also, maybe go for a run once in a while.

```
# This is a comment, R will ignore everything on this line

1 + 1 # You can put comments on the same line after code

## [1] 2

# Note to self: Put on an Evil Dead marathon with Bloody Mary's
```

Figure 8: 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.

Code formatting guidelines

While it's important that R understands your code, it's equally as important that other humans (including your future self) understand your code. For this reason, it's important to make appropriate use of spaces, line breaks, and comments. For example, I include spaces between arithmetic operators (like =, + and -) and after commas (which we'll get to later). Personally, this makes code much easier for me to follow. 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

Assigning objects with the <- assignment

So far so good, you can use R as a simple calculator. Now, let's do our first *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.

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` and `March2015Group1OnlyMales` will give you carpel tunnel syndrome.

To do an assignment, we use the almighty <- operator.

```
new.object <- something
```

To assign something to a new object (or to update an existing object), use the notation `new.object <- something`, where `new.object` is the new (or updated) object, and `something` is whatever you want to store in `new.object`.

a <- b

Let's start by creating a very simple object called `a` and assigning the value of 100 to it:

```
a <- 100 # Assign the value of 100 to a new object called a
```

Now, anytime we want to refer to the content of the object `a`, we can just type it. You may have noticed that when you run the code

above, R won't actually show you the value of a. If you want to see the value, you need to call the object by just executing its name:

```
a

## [1] 100
```

Now, R will print the value of a (in this case 100) to the console.

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

How to name objects

You can create object names using any combination of letters and a few special characters (like .). However, you can't start the name of an object with a number, or use spaces in an object name. Let's create some new objects with meaningful 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 which contain either spaces, start with numbers, or have invalid characters:

```
a b <- 50 # Can't have spaces
5a <- 50 # Can't start a name with a number
a! <- 5 0# Can't have an "!" in the object name
```

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.

R is case-sensitive!

Like English, R is case-sensitive. This means that 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:

```
Plunder <- 1
plunder <- 100
PLUNDER <- 5

Plunder
```

Figure 9: Like a text message, you should probably watch your use of capitalization in R.

```
## [1] 1

plunder

## [1] 100

PLUNDER

## [1] 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.

Changing and combining objects

Once you've defined an object, you can change it by reassigning it, or combine it with other objects using basic arithmetic.

```
# Set a to 100, then add 100 to it
a <- 100
a <- a + 100
a

## [1] 200

# Set b to a times 5
b <- a * 5
b

## [1] 1000

# Now let's set a to 1
a <- 1
a

## [1] 1
```

Let's define an object called `blackpearl.usd` which has the global revenue of Pirates of the Caribbean 1 "Curse of the Black Pearl" in U.S. dollars. A quick Google search showed me that the revenue was $634,954,103:

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

Now, my German friends might want to know how much this is in Euros. Let's create a new object called `blackpearl.eur` which

Tip: To change (reassign) an object, you have to use the <- operator - If you don't, the object won't change. In the following example, the object a does not change because we did not reassign it with <- :

```
a <- 1
a + 99 # this will just print a + 99

## [1] 100

a # a is still just 1

## [1] 1
```

Now, let's fix the code by reassigning a to a + 99

```
a <- 1
```

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
```

Now, let's see how much more Pirates of the Caribbean 2 "Dead Man's Chest" made compared to "Curse of the Black Pearl" which made $1,066,215,812 (that wasn't a mistype, the freaking movie made over a billion dollars). First I'll create a new object called "deadman.usd" with the revenue of "Dead Man's Chest". Then, I'll divide deadman.usd by blackpearl.usd

```
deadman.usd <- 1066215812
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.

Numbers versus characters

There are two classes of objects in R, numeric and character (also known as strings). So far, we've been dealing mostly with numeric objects. To define a character object, use quotation marks.

Here are some examples of character objects:

```
actor <- "Johnney Depp"
movie <- "Black Pearl"
```

As you can probably guess, R treats numeric and character objects very differently. For example, you can't do arithmetic operations on characters. R also has many special functions for dealing with characters that don't apply to numbers. We'll get to these functions in a later chapter. For now, just know that numeric and character objects are treated very differently.

Note that even when you are defining a character object, the actual name of the object should *not* have quotation marks. For example, don't write *"actor" <- "Johnny Depp"*.

To refer to an object, make sure you don't use quotation marks, even if the object is a character. Otherwise, R will think you're referring to a new string:

```
actor <- "Johnney Depp"
"actor" # Just a string, not the object

## [1] "actor"

actor

## [1] "Johnney Depp"
```

Scalers and vectors

So far, we've been dealing with singular objects. That is, single numbers, or single strings. These are called *scalers* in R. But how does R store lists of numbers - like the revenues of all the Pirates of the Caribbean movies)? These are stored in objects called *vectors*.

scalars

Actually you're used to scalers by now. A **scalar** is just a single value. As you know, a scalar can either be *numeric* or *character*.

Here are examples of numeric scalers:

```
a <- 100
b <- 3.14
c <- 3 / 100
```

Here are examples of character scalers:

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

Vectors

Now let's move onto *vectors*. A **vector** 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. 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 is a table of the most common ones:

```
# 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")
```

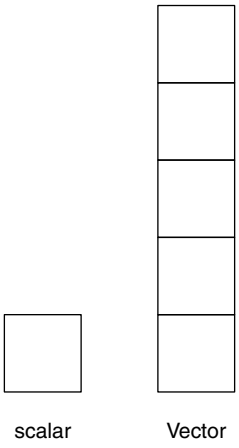


Figure 10: Visual depiction of a scalar and vector. Deep shit. Wait until we get to matrices - you're going to lose it.

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]

Let's go through each function in detail:

`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".

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 `my.vec` containing the integers from 1 to 5.

```
my.vec <- c(1, 2, 3, 4, 5)
```

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

```
my.vec

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

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

You can also create character vectors by using the `c()` function to combine character scalers. Here are some examples of character vectors:

```
str.vec1 <- c("I", "don't", "like", "Android", "swords")
str.vec2 <- c("this", "is", "not", "a", "pipe")
```

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:

```
one.to.five <- c(1, 2, 3, 4, 5)
six.to.ten <- c(6, 7, 8, 9, 10)
one.to.ten <- c(a, b)
one.to.ten

## [1] 100.00   3.14
```

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.

```
movie <- "Pirates of the Carribean"
revenue <- 634954111
c(movie, revenue) # Result is a string vector

## [1] "Pirates of the Carribean" "634954111"
```

Generating numeric vectors

While the c() operator is the most straightforward way to create a vector, it's also one of the most tedious. 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. Instead, 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 creates 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:b
a The start of the sequence
b The end of the sequence

Here are some examples of the `a:b` function in action:

```
1:10 # 1 to 10

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

10:1 # 10 to 1

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

20.1:30.1 # From 20.1 to 30.1

##  [1] 20.1 21.1 22.1 23.1 24.1 25.1 26.1 27.1 28.1 29.1 30.1
```

seq(from, to, by, length.out)

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, with `seq()`, you can specify either the size of the steps between numbers, or the total length of the sequence:

seq(from, to, by, length.out)

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`)

seq(from, to, by) - Creates a sequence between two numbers in steps that you specify.
`from`: The starting value
`to`: The ending value
`by`: The step size between `begin` and `end`

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

seq(from = 20, to = 0, by = -2)

##  [1] 20 18 16 14 12 10  8  6  4  2  0
```

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

seq(from = 0, to = 100, length.out = 2)

## [1]   0 100
```

rep(x, times, each)

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

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

rep(x, times, each) - Repeats the numbers in x in a manner you specify
`times`: The number of times the vector should be repeated
each: The number of times you want to repeat each element in the vector.

For example, let's say you are are getting a batch of 10 new pirates on your ship, 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.. Let's create this vector using `rep()`

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

You can also use `rep()` to repeat vectors of character vectors. Let's repeat the previous example, but instead of using the numbers 1, 2, 3, we'll use the names of the actual jobs

```
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"
```

length()

length(x)

Once you have a vector, you may want to know how long it is. Don't stare at your computer screen and count the elements one by one! Instead, use `length()` function:

```
length(1:10)

## [1] 10

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

## [1] 20

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

## [1] 6

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

## [1] 1
```

Test your R might!

1. 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.

2. 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 a character object called `plank.list` with the name of the pirate thief.

3. 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
```

4. 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()`.

5. Create the vector [2.1, 4.1, 6.1, 8.1] in two ways, once using `c()` and once using `seq()`

6. Create the vector [0, 5, 10, 15] in 3 ways: using `c()`, `seq()` with a `by` argument, and `seq()` with a `length.out` argument.

7. Create the vector [101, 102, 103, 200, 205, 210, 1000, 1100, 1200] using a combination of the `c()` and `seq()` functions

8. 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).

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

Additional Tips

1. If you want to run a single line of code, you don't need to highlight anything. Instead, click on the line you want to run and use the "Run" hot-key (`command + enter` on Mac), or click the "Run" button. R will execute just that line - and automatically move the cursor to the next line. To run the next line, just hit the hot-key again. This is a great way to test your code line-by-line.

2. To get more tips on how good coding techniques, check out the R style guide at http://adv-r.had.co.nz/Style.html. For great blog articles on R, check out http://www.r-bloggers.com/

3. If you need to enter a lot of numeric data into R by hand you might want to use the `scan()` function. This function allows you to easily enter data using 10-key typing on a number pad. To do this, run the code `scan()` and then enter the data number by number. When you are finished, R will then print the appropriate code to store the data into a vector.

4. You can run several lines of code in one line by separating the code with the ; key. For example, the following two chunks of code are the same:

```
a <- 1
b <- 14
c <- 67
```

```
a <- 1 ; b <- 14 ; c <- 67
```

However, I recommend you use the ; key sparingly (if at all). If you get in the habit of trying to cram several lines of code in one line, your code will get cluttered and difficult to understand.

3: Vector arithmetic and descriptive statistics

Chapter Goals

1. Do simple math operations on vectors
2. Learn functions for basic descriptive statistics: mean(), median(), sd(), var(), min(), max()

Arithmetic operations on vectors

So far, you know how to do You can 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.

Basic math with vectors and scalers

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 want to add 100 to every element in a vector, you can just treat them both like scalers. Let's create a vector with the integers from 1 to 10 and add 100 to each element:

```
a <- 1:10
a + 100

##  [1] 101 102 103 104 105 106 107 108 109 110
```

The result is a vector where the first element is 1 + 100, the second element is 2 + 100 (etc.). Of course, this doesn't only work with addition...oh no. Let's try division, multiplication, and exponents:

```
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? For example, 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 <- pies + cookies
total

## [1]  73  46  42 210  64
```

As this example shows, if you do an operation on two vectors, R will try to apply the operation between the vectors by each item. For example, if you add two vectors of equal length (like we did above), R will create a new vector of the same length where each element is the sum of the elements in the previous vector.

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
```

If you try to do basic operations on two vectors of unequal length, R will try to increase the length of the shorter vector by repeating it. For example, if you try to add [1, 2] with [1, 2, 3, 4, 5, 6], you get the following result

```
a <- c(0, 100)
b <- c(1, 2, 3, 4, 5, 6)
a + b

## [1]   1 102   3 104   5 106
```

```
ab.prod

## [1]  1  4  9 16 25
```

Summary statistics for Continuous data

Ok, now that we can generate some data, let's learn the basic descriptive statistics functions. We'll focus on the most common ones for numerical analyses. Each of the following functions takes a vector as an argument, and returns a scalar as a result.

Common Descriptive Statistics

`mean(x)`

The arithmetic mean of a vector x

`median(x)`

The median of a 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 vector x.

`min(x), max(x)`

The minimum and maximum values of a vector x

`quantile(x, p)`

The pth sample quantile of a 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:

```
mean(tattoos)

## [1] 24.1

median(tattoos)

## [1] 9

sd(tattoos)

## [1] 31.32074

min(tattoos)

## [1] 2

max(tattoos)

## [1] 100
```

Watch out for NA values!

One important point about the descriptive statistics functions is that most (if not all) of them as a default 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:

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

```
mean(c(1, 5, NA, 2))

## [1] NA
```

To tell a descriptive statistic function to ignore missing (NA) values, include the argument `na.rm = T` in the function:

```
mean(c(1, 5, NA, 2), na.rm = T)

## [1] 2.666667
```

Now, the function will ignore NA and calculate the mean of the non-missing values. 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 large existing datasets that may contain 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
```

Other helpful vector functions

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

Other helpful numeric 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).

Standardizing (z-score) variables

During one of your morning strolls on your ship's deck, you see two pieces of paper with the titles: "Mugs of Grogg" and "Climbing" respectively. On each piece, you see the following numbers:

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

Now you know what was keeping you up last night: it turns out your crew of 5 pirates had two contests last night. In one, the pirates drank as many mugs of grogg as they could in 5 minutes. In the second, they saw how many feet of rope the pirate could climb in an hour. Now you're curious how well each pirate did and which performances were the most spectacular. 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 can use *standardization*. Put simply, standardizing is a common method to put all vectors of numbers on a similar scale. Specifically, it requires subtracting the mean from a set of numbers, and then dividing the numbers by their standard deviation. Once you've standardized a set of numbers, you can interpret a value of x as being 'x standard deviations away from the mean.'

Let's use standardization to create new 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, 1)

## [1]  1.4  0.5 -1.1  0.0 -0.8

round(climbing.z, 1)

## [1] -1.2  0.5  0.2 -0.8  1.3
```

It looks like we we have 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 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
```

Here's an easy function for standardization:

```
standardize <- function(x) {

  return((x - mean(x)) / sd(x))

}
```

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.

Summary Statistics for Discrete data

Next, we'll move on to common summary statistics for discrete data. Discrete data are those like gender, occupation, or favorite pirate movie that only allow for a finite (or at least, plausibly finite) set of responses.

Discrete data summary statistics

`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 start 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(x): Gives you all unique values in a vector, ignoring the number of times 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(x): Gives you all unique values in a vector and tells you how often each value 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
```

Additional Tips

Test your R Might!

1. Create a vector that shows the square root of the integers from 1 to 10.
2. Create a vector that shows the powers of 2 from 1 to 10. That is, the first element should be 2^1, $these second elements hould be 2^2 (etc.)$

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)
```

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

4. Create a new vector called `difference` that shows how much more treasure Renata found while drinking grogg than when she didn't drink grogg.

5. What was the mean, median, and standard deviation of the difference?

4: Indexing and comparing vectors

Chapter Goals:

1. Use brackets [] and logical vectors to index vectors
2. Combine indexing with descriptive statistics
3. Learn indexing functions which(), sort()
4. Vector discrete summary functions table() and unique()
5. Set functions: intersect(), union(), setdiff(),

Indexing vectors with brackets

Indexing with brackets [ ]
To get the ith value of a vector called vec, use the bracket notation vec[i]

When we have a vector of data, we will frequently want to access specific values of a vector. These might be values in a specific location in the vector (i.e.; the fifth element) or based on some criteria (i.e.; all values greater than 0). For example, let's say we did a survey of pirates and asked each pirate two questions: "What is your favorite board game?" and "How long is your beard (in cm)?". We can represent these data in two vectors:

```
fav.game <- c("Pir of Catan", "Pir of Catan", "Piratopoly",
              "Piratopoly", "Pir of Catan")

beard.length <- c(30, 24, 0, 40, 15)
```

As you can see, we have two dominant board games here: Pirates of Catan and Piratopoly. Now, what if we wanted to calculate summary statistics, on beard length (like the mean, median, or standard deviation) separately for pirates whose favorite board game is Pirates of Catan and for those whose favorite game is Piratopoly? Unfortunately we can't simply do this by applying the summary statistic function to the `beard.length` vector. We need to find some way of telling R to *index* the `beard.length` vector based on the different values of the `fav.game` vector.

In this chapter we'll cover different ways of indexing vectors. Before we start with the basics (and because I'm sure you're dying to

know if a pirate's beard length depends on his favorite board game), here's how we can use indexing to answer our previous question:

```
a <- beard.length[fav.game == "Pir of Catan"]
b <- beard.length[fav.game == "Piratopoly"]

mean(a)

## [1] 23

mean(b)

## [1] 20
```

Looks like the mean beard length of pirates whose favorite game is Pirates of Catan is 23, compared to 20 for those who prefer Piratopoly. Who would have guessed.

vector[index]

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 data[num.index], where `data` is the data vector, and `num.index` is a vector of index values. For example, to get the first value in a vector, you'd write `data[1]`. To get the first, second, and third value, you can either type `data[c(1, 2, 3)` or `data[1:3]`.

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

```
tattoos <- c(0, 50, 2, 39, 9, 20, 17, 8, 10, 100)
tattoos[1] # First element of tattoos

## [1] 0

tattoos[1:5] # 1-5 elements of tattoos

## [1]  0 50  2 39  9
```

If you have defined an object that is a vector of integers, you can then index a variable using that vector. For example, let's define an object called `index` and use this object to index our data vector:

```
get.these.values <- 6:10
tattoos[get.these.values] # Indexing with a named object

## [1]  20  17   8  10 100
```

As you gain more experience with R, you'll realize that there are many ways to program the same result. The choice of which code you use comes down to a delicate balance of readability (How easily can your future self, and other people, understand what the code is doing?), simplicity (How many lines of code are necessary?), and processing speed (How quickly will R complete the task?).

Creating logical vectors

Another way to index data vectors is with logical vectors. A logical vector is a vector that only contains TRUE and FALSE values. If you index a vector with a logical vector (of the same length), you will only receive the values for which the index is TRUE.

You can create a logical vector by using the comparison operators in Figure 11.

Let's start by creating single scalar logical values so you can see how they work. If you apply a comparison operator to a scalar, R will return a single logical value of TRUE or FALSE. Let's see if 3 truly equals 3 and if 3 is really not greater than 5.

```
3 == 3

## [1] TRUE

3 > 5

## [1] FALSE
```

The negation operator ! meaning NOT. To use it, place the statement you are testing in parentheses, and place the ! operator before it:

```
pirate <- "david"
pirate == "jack"

## [1] FALSE

!(pirate == "jack")

## [1] TRUE

!(2 == 4)

## [1] TRUE
```

You can also get random values from a vector by indexing a vector with the `sample()` function. Let's get 3 random values from the `tattoo` vector in 2 steps. First, we'll create 3 random indexing values using `sample()`. Second, we'll index the `tattoo` object with the indexing values we generated in the first step.

```
rand.values <- sample(x = 1:length(tattoos),  # Step
                      size = 3,
                      replace = F)

tattoos[rand.values] # Step 2: Index tattoo with ran

## [1] 17 50  2
```

The result of our indexing is 3 randomly selected values from the `tattoos` vector. Of course, we also could have done the same thing in one step by just entering `tattoos` as an argument to `sample()` like this:

```
sample(x = tattoos, size = 3, replace = F)

## [1]  8 10 50
```

```
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 11: Comparison operators in R

```
# Create blank plot with no margins
par(mar = rep(0, 4))
plot(1, xlim = c(0, 1), ylim = c(0, 13),
     bty = "n", xlab = "", ylab = "", main = "",
     type = "n", xaxt = "n", yaxt = "n")

# Add Main title
text(.5, 12.5, "log.vec <- data.vec > 0", cex = 2)

# Data vector
text(.3, 11.1, "data.vec", font = 2, cex = 1.6)
data.vec <- c(2, 7, -1, 5, -9, -2, 3, 0, 2, -2)
text(rep(.3, 10), 10:1, data.vec, cex = 1.6)
rect(.25, .5, .35, 10.5)
segments(rep(.25, 9), seq(1.5, 9.5, 1),
         rep(.35, 9), seq(1.5, 9.5, 1), lty = 2)


# Comparisons
text(.5, 11.1, "test", cex = 1.6, col = gray(.5))
text(rep(.5, 10), 1:10, "> 0", col = gray(.5))

# Logical vector
```

In addition to using single comparison operators, you can combine multiple logical comparisons using the OR | and AND & commands. The OR command will return TRUE if any of the values in the set is TRUE, while the AND command will only return TRUE if all of the values in the set are TRUE.

```
(1 < 3) # Is 1 less than 3?

## [1] TRUE

(4 < 2) # Is 4 less than 2?

## [1] FALSE

(1 < 3) & (4 < 2) # Is 1 less than 3 and is 4 less than 2?

## [1] FALSE

(1 < 3) | (4 < 2) # Is 1 less than 3 OR is 4 less than 2?

## [1] TRUE
```

If you apply a comparison operator between a scalar and a vector, R will return a logical vector of length equal to the length of the vector. For example, let's compare a vector of integers from 1 to 10 to a scalar value of three and look at the output:

```
1:10 == 3 # Are the values equal to 3?

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

Let's look at the outputs above: for each value of the object vec, R performs the comparison == 3. Because only the third element of the vector is equal to 3, R returns the value FALSE for all values except the third one.

You can also compare two vectors of equal length and obtain a single logical vector as a result. For example, let's say we have two data vectors (data.1 and data.2) and we want a logical vector telling us which values of the two data vectors are equal. We can do this by just executing `data.1 == data.2`

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

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

`x %in% y`

One very important function for creating logical indices is `%in%`. This function looks a bit different from other functions because it doesn't follow the typical format of function(x, y). Instead, you place the function %in% between its arguments. When you execute x %in% y, R will evaluate, for each element in the vector x, if it is in the vector y. For example. Let's create several vectors x and y and use the %in% function to text whether or not the elements of x are in y:

```
1 %in% c(1, 2, 3, 4, 5)

## [1] TRUE
```

In this example, R returns a single value of TRUE because it found the value of 1 in the second vector. However, you can also apply the %in% function to a vector x that is longer than 1. When you do this, the %in% function will return a vector equal to the length of x. Now, let's try an example where we test whether each of several values are in a second set:

```
c(1, 2, 3, 77, 88, 99) %in% c(1, 2, 3, 4, 5)

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

In this example R checked, for each of the values in the first vector if it was in the second vector (`c(1, 2, 3, 4, 5)`). Because only the first three values (1, 2 and 3) were in the second vector, R returns a vector with 3 TRUE values and 3 FALSE values.

The %in% function is very handy for seeing which values in a vector are valid according to a criteria you specify. For example, imagine you conducted a survey where you asked 10 different pirates how many siblings they had and received the following responses:

```
siblings <- c(3, 2, 0, -5, 0, -20, 2, 3, 1, -200)
```

Of course, the only valid answers to this question should be 0, 1, 2, ... up to a maximum of say 20; but some of these values appear to be invalid (that is, negative). Let's use the %in% function to see which values in the survey are valid. We'll create a vector called `valid.responses` that represents all possible valid answers to the question (we'll limit the number of siblings to 20). We'll then use %in% to create a logical vector indicating which responses were valid.

```
siblings <- c(3, 2, 0, -5, 0, -20, 2, 3, 1, -200)
valid.responses <- seq(0, 20, 1)
siblings %in% valid.responses

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

Because the fourth, sixth, and tenth values were not valid (they were negative), the final logical vector gives us FALSE values at those index values, and TRUE values for all others.

Indexing data with logical vectors

Once we have a logical vector, we can use that vector as an indexing vector. That is, you can use it to select values of a vector that satisfy some criteria you specify. To do this, you create a logical vector containing TRUE and FALSE values. If you then index a data vector (with the same length as the logical vector), R will return the values of the data vector for all TRUE values of the logical vector. See Figure to see visually how this works.

For example, let's say that we have the following set of data

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

Now, let's say that we want to access just the data points that are less than 10. We'll start by first creating a logical indexing vector that tells us whether each value is less than 1. Then, we'll index the original data vector using this logical vector:

```
log.vec <- tattoos < 10 # Step 1: Create logical vector
tattoos[log.vec] # Step 2: Index the original data by the logical vector

## [1] 4 2 4 4 8
```

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()

`which(log.vec)`

If you apply the function `which()` to a logical vector, R will tell you which values of the index are TRUE. For example, let's create a logical vector and then see which index values are TRUE

```
log.vec <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
which(log.vec)

## [1] 1 2 4
```

By using the `which()` function, we know that the first, second, and fourth elements of the logical vector are TRUE.

Let's take the example of comparing the treasure chest finding ability of 10 pirates. In each of two years - 2014 and 2015 - I measured

```
# Create blank plot with no margins
par(mar = rep(0, 4))
plot(1, xlim = c(0, 1), ylim = c(0, 13),
     bty = "n", xlab = "", ylab = "", main = "",
     type = "n", xaxt = "n", yaxt = "n")

# Add Main title
text(.5, 12.5, "output.vec <- data.vec[log.vec]", cex = 2)

# Data vector
text(.2, 11.1, "data.vec", font = 2, cex = 1.6)
data.vec <- c(2, 7, -1, 5, -9, -2, 3, 0, 2, -2)
text(rep(.2, 10), 10:1, data.vec, cex = 1.6)
rect(.15, .5, .25, 10.5)
segments(rep(.15, 9), seq(1.5, 9.5, 1),
         rep(.25, 9), seq(1.5, 9.5, 1), lty = 2)
text(rep(.12, 10), 10:1, 1:10, cex = .8)

# Comparisons
text(rep(.32, 10), 1:10, "> 0", col = gray(.5))

# Logical vector
text(.5, 11.1, "log.vec", font = 2, cex = 1.6)
index.text <- rep("FALSE", 10)
index.text[data.vec > 0] <- "TRUE"
col.vec <- rep("red", 10)
col.vec[data.vec > 0] <- "blue"
text(rep(.5, 10), 10:1,
     index.text,
     col = col.vec, cex = 1.6
     )
rect(.4, .5, .6, 10.5)
segments(rep(.4, 9), seq(1.5, 9.5, 1),
         rep(.6, 9), seq(1.5, 9.5, 1), lty = 2)

# Output vector
text(.8, 11.1, "output.vec", font = 2, cex = 1.6)
output.text <- data.vec[data.vec > 0]
text(rep(.8, 5), 7:3, output.text, cex = 1.6)
rect(.75, 2.5, .85, 7.5)
segments(rep(.75, 9), seq(3.5, 6.5, 1),
         rep(.85, 9), seq(3.5, 6.5, 1), lty = 2)
text(rep(.88, 5), 7:3, which(data.vec > 0), cex = .8)

# Arrows connecting log.vec to output.vec
arrows(rep(.6, 5),
       11 - which(data.vec > 0),
       rep(.73, 5),
       7:3, lwd = .5, length = .15
       )
```

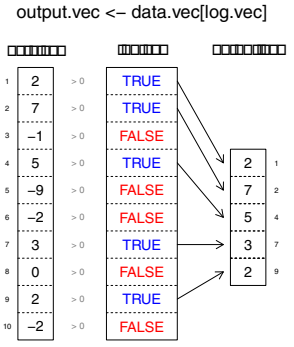


Figure 13: A visual representation of how indexing with logical vectors works in R. When you apply a logical vector (a vector containing only TRUE and FALSE values) to a data vector, R will return the the values in the data vector where the logical vector is TRUE.

how many chests 10 pirates found over the entire year. I recorded these values in two vectors, where the first value of each vector corresponds to the first pirate, and the last value corresponds to the last pirate:

```
pirate.names <- c("Andrew", "Heidi", "Madisen", "Becki", "Jack Dyanamite")
chests.2014 <- c(0, 10, 1, 2, 5)
chests.2015 <- c(0, 6, 3, 0, 20)
```

Ok, so let's see which pirates improved their chest finding ability. I'll start by finding the index values where the number of chests found increased between the two years

```
improve.log <- chests.2015 > chests.2014 # create logical vector
improve.log # print values

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

If I want to know the index values of the pirates who improved, I can use the `which()` function. The which function will tell me the index of each TRUE value in a logical vector:

```
which(improve.log)

## [1] 3 5
```

This vector tells us that the 3rd and 5th pirates found more chests in 2015 than 2014. Now I can use this index value to figure out the names of those pirates:

```
pirate.names[which(improve.log)]

## [1] "Madisen"        "Jack Dyanamite"
```

Because you can index vectors with logical vectors, I could get the same results by just indexing `pirate.names` with `improve.log`.

```
pirate.names[improve.log]

## [1] "Madisen"        "Jack Dyanamite"
```

For this example, the `which()` command was unnecessary, but it's important to understand the logic of both methods.

Additional helpful vector functions

Here are some other functions you might find useful when dealing with vectors:

Other Helpful Vector Functions

`length(x)`

The length of a vector

`sort(x)`

Sort a vector x. Add the argument `decreasing = T` to sort in decreasing order.

`rev(x)`

Reverse the order of a vector x

`rank(x)`

Returns the sample ranks of the values in a vector. Ties (i.e.; equal values) and missing values can be handled in several ways using the `ties.method` argument.

Once you have a vector of data, you may want to sort it in order to see, for example, the largest and smallest values. You can do this using the `sort()` function. Let's look back on my summer joke data and sort the results:

```
tattoos <- c(4, 50, 2, 39, 4, 20, 4, 8, 10, 100)
sort(tattoos, decreasing = T) # Sort decreasing

##  [1] 100  50  39  20  10   8   4   4   4   2

sort(tattoos, decreasing = F) # Sort increasing

##  [1]   2   4   4   4   8  10  20  39  50 100
```

You'll notice that the sort function has an argument `decreasing` which you can set to TRUE or FALSE.

Set Functions

R contains many functions that allow you to compare two sets (vectors) of data. See margin Figure for a visual depiction. Here are the most common ones:

Set Functions

`union(x, y)`

Tells you all unique values included in *either* the vector x or y.

`intersect(x, y)`

Tells you all values common in *both* the vectors x and y.

`setdiff(x)`

Tells you which values are in the vector x but *not* in the vector y. Keep in mind that `setdiff(x, y)` is *not* the same as `setdiffy, x`!

`setequal(x)`

Returns TRUE of the two vectors x and y are identical (ignoring order) and FALSE if they are not identical.

```
require("plotrix")

## Loading required package: plotrix
## Warning: package 'plotrix' was built under R version
3.2.3

require("RColorBrewer")

# Transparent() is a function that makes transparent colors
Transparent <- function(orig.col = "red",
                        trans.val = 1,
                        maxColorValue = 255) {

  orig.col <- col2rgb(orig.col)

  final.col <- rgb(orig.col[1],
                   orig.col[2],
                   orig.col[3],
                   alpha = trans.val * 255,
                   maxColorValue = maxColorValue)

  return(final.col)
}


color.vec <- brewer.pal(12, "Set3")
par(mar = rep(0, 4))
plot(1, xlim = c(0, 1), ylim = c(0, 1),
     bty = "n", xlab = "", ylab = "", main = "",
     type = "n", xaxt = "n", yaxt = "n")

draw.circle(x = .35, y = .5,
            radius = .35,
            col = Transparent(color.vec[4], .3), lwd = 2)

draw.circle(x = .65, y = .5,
            radius = .35, col = Transparent(color.vec[5], .3),
            lty = 2, lwd = 2)

text(.35, .1, "Set X", cex = 1.5)
text(.65, .1, "Set Y", cex = 1.5)

text(.5, .5, "intersect(x, y)")
text(.15, .5, "setdiff(x, y)")
text(.85, .5, "setdiff(y, x)")
text(.5, .9, "union(x, y)")
```

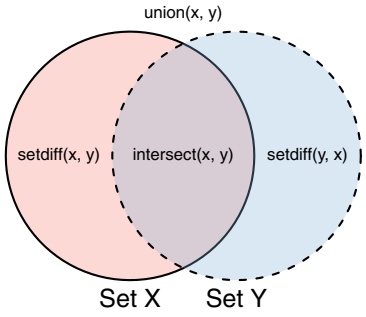


Figure 14: Common set functions in R.

Using indexing to remove specific values of a vector

Sometimes you might want to remove values of a vector before performing some analyses. This might be because some of the values are invalid or just not values that you want to include in your analyses. For example, let's say you asked 7 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)
```

As you can see, we have some invalid values (999 and -2) in this vector. We can use logical indexing to create a new vector called `happy.valid` that only contains values 1 through 5.

```
valid.log <- happy %in% c(1, 2, 3, 4, 5)
happy.valid <- happy[valid.log]
happy.valid

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

As you can see, the new vector `happy.valid` only contains values from the original vector that are integers from 1 to 5.

R has special functions for testing whether or not values in a dataset are either missing (or infinite). Here are some you can 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)]`

You can use these functions to generate logical indices for indexing. For example, let's say you had a vector of data with several missing values. To create a new vector of data that does not contain the original NA values, we can index the original `data` vector with `is.finite(data)`:

```
data <- c(5, 2, NA, 3, NA, 10, NA)
data.finite <- data[is.finite(data)]
data.finite

## [1]  5  2  3 10
```

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

Many (if not all) R functions that take numeric data as inputs 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.

Let's use this logic to see how many of the integers from 1 to 100 are greater than 0, 50, and 100:

```
sum(1:100 > 0) # How many values in 1:100 are greater than 0?

## [1] 100

sum(1:100 > 50) # How many values in 1:100 are greater than 50?

## [1] 50

sum(1:100 > 100) # How many values in 1:100 are greater than 50?

## [1] 0
```

These results should make sense: every value from 1:100 is greater than 0, 50 are greater than 50, and non are greater than 100. Now, let's do the same thing but calculate percentages instead of counts using `mean()` instead of `sum()`:

```
mean(1:100 > 0) # How many values in 1:100 are greater than 0?

## [1] 1

mean(1:100 > 50) # How many values in 1:100 are greater than 50?

## [1] 0.5

mean(1:100 > 100) # How many values in 1:100 are greater than 50?

## [1] 0
```

So far so good, now let's try this on our tattoo data:

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

Let's see see how many of these 10 pirates have more than 10 tattoos. We'll do this in two steps; First, we'll create a logical vector indicating which values are greater than 10. Second, we'll take the sum of this logical vector. This will tell us how many `TRUE` values there are in the logical vector:

```
log.vec <- tattoos > 10 # Step 1: Which values are > 10?
sum(log.vec) # Step 2: How many TRUE values are there?

## [1] 4
```

Looks like 4 pirates have more than 10 tattoos. Now, let's test what percent of pirates have 5 tattoos or less. We'll do this by first creating the logical vector, and then calculating the `mean()` of this vector. We can do this because the mean of a vector of 0s and 1s is identical to the percentage of 1s:

```
log.vec <- tattoos <= 5 # Step 1: Which values are <= 5?
mean(log.vec) # Step 2: What percent of values are TRUE?

## [1] 0.4
```

Looks like 40% of pirates have 5 tattoos or less.

To see what percentage of values are TRUE in a logical vector, just take the mean of the vector. For example, the command `mean(c(-1, -2, 1, 1) > 0)` will return `0.50`, telling you that half of the values are positive.

Additional Tips

- If you have a vector of values and you want to know which values are duplicates of previous values, you can use the `duplicated` function. This function will go through the vector from beginning to end and tag the first unique instance of a value as TRUE and all repeated instances of a value as FALSE:

```
vec <- c("a", "b", "a", "a", "c")
duplicated(c("a", "b", "a", "a", "c"))

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

If you want to remove duplicated values from a vector, you can just index the vector by `!duplicated`:

```
vec[!duplicated(vec)]

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

However, you can do the same thing with `unique()`!

A worked example - Chicken Weights

A farmer is testing the effectiveness of three different diets on the weight gain of chickens. When they are born, 50 chicks are randomly assigned to one of 4 diets. Over several time periods, the farmer weighs each chicken. These data are contained in the dataset `ChickWeight`. Because the data are stored in a dataframe, which we haven't learned yet, we'll convert the four columns in the dataset to vectors as follows:

```
weights <- ChickWeight$weight
time <- ChickWeight$Time
chick <- as.numeric(paste(ChickWeight$Chick))
diet <- as.numeric(ChickWeight$Diet)
```

Let's answer 5 questions with these vectors:

1. What are the first 10 elements of the `weights` vector and the last 10 elements of the `weights` vector?

```
weights[1:10]

##  [1]  42  51  59  64  76  93 106 125 149 171

weights[(length(weights) - 9):length(weights)]

##  [1]  67  84 105 122 155 175 205 234 264 264
```

To answer the second question, I used the `length()` function to index index `weights` to go from 9 elements *before* the end of the vector, to the end of the vector.

2. Which chicks were given diets 1 and 2?

```
unique(chick[diet == 1])

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

unique(chick[diet == 2])

##  [1] 21 22 23 24 25 26 27 28 29 30
```

3. What was the mean weight across all time periods separately for diets 3 and 4?

```
mean(weights[diet == 3])

## [1] 142.95

mean(weights[diet == 4])

## [1] 135.2627
```

First, I indexed the `weights` vector with a logical vector created from from `diet`. I then calculated the mean of this indexed vector.

4. What was the standard deviation of weights for diets 1 and 2 at time < 10?

```
sd(weights[diet <= 2 & time < 10])

## [1] 17.2321
```

5. What was the median weight for chicks 10, 20, and 30 for time periods greater than 10?

```
median(weights[chick %in% c(10, 20, 30) & time > 10])

## [1] 115
```

6. Which chicks did not make it until the final time period?

This one is a bit tricky. First, I need a vector of all chicks in the study (`all.chicks`). Next, I need a vector of all chicks that survived to the last time point (`surviving.chicks`). Third, I need to test, for each chick, whether they are present in the vector of surviving chicks (`survived.log`). Finally, I index the vector of all chicks where the logical index is FALSE (because we want chicks that did not survive).

```
# Step 1: Create a vector of all chicks (all.chicks)
all.chicks <- sort(unique(chick))

#Step 2: Create a vector of all chicks that survive until the end (surviving.chicks)
surviving.chicks <- sort(unique(chick[time == max(time)]))

# Step 3: For each chick, see if it is present in the vector of surviving chicks
survived.log <- all.chicks %in% surviving.chicks

# Step 4: Index the vector of all chicks by the logical vector
all.chicks[survived.log == FALSE]

## [1]  8 15 16 18 44
```

Test your R might!

The following vour vectors give data about 10 movies.

```
m.names <- c("Baramgwa hamjje sarajida", "Sleepless in Seattle", "The Water Diviner",
"Fly Away Home", "The Three Musketeers", "Candyman: Farewell to Flesh",
"Honey I Threw our Kid off the Plank", "Kingsman: The Secret Service", "Ajab Prem Ki Ghazab Kahani",
"A Bug's Life")

boxoffice <- c(28686545, 218076024, 30864649, 35870837, 50375628, 13899536,
58662452, 404561724, 15906411, 363089431)

genre <- c("Action", "Romantic Comedy", "Drama", "Drama", "Adventure",
"Horror", "Comedy", "Action", "Comedy", "Adventure")

time <- c(121, 100, 112, NA, NA, NA, NA, 129, NA, 96)

rating <- c(NA, "PG", "R", "PG", "PG", "R", "PG", "R", NA, "G")
```

1. What is the name of the 10th movie?
2. What are the genres of the first 4 movies?
3. What are the names of every second movie? (that is, the 2nd, 4th, 6th, 8th, and 10th).
4. Some joker changed the name of the Pixar film "A Bug's Life" to "A Pirate's Life." Please correct the name of this movie.
5. Create a new vector called âĂIJboxoffice.millionsâĂİ that has the box-office values in millions of dollars. For example, a value

of 1000000 in the original boxoffice vector should be 1 in boxoffice.millions

6. What is the mean, median, and standard deviation of the box-office totals of all movies?

7. What where the different movie genres and how many movies are there of each genre? (hint: use table())

8. How many movies were Dramas? (hint: don't use table(), use sum())

9. What was the box-office total, genre, running time, and rating of "A Bug's Life"?. Do this once using numerical indexing and once using logical indexing.

10. What were the names of the movies that made less than $30 Million dollars AND were Comedies?

5: Matrices and Data Frames

Chapter Goals

1. Learn about the matrix and dataframe data objects
2. Create matrices with matrix(), cbind(), and data.frame()
3. Index matrices/dataframes with brackets [], and $
4. Use matrix/dataframe functions dim(), nrow(), ncol(), head(), tail()
5. Import datasets

Creating matrices and dataframes

By now, you should be comfortable with scalars and vectors. Next, we'll cover the next two most common data objects in R, **matrices** and **dataframes**

Matrices and dataframes are both two dimensional objects that contain rows and columns. Really, they're just like spreadsheets in Excel. Each matrix or dataframe contains a certain number of rows (call that number m) and columns (n). You can think of a matrix as a combination of n vectors, where each vector has a length of m. See Figure 15 to see the difference.

You can use several functions in R to create matrices and dataframes. In the next sections we'll cover the most common ones.

```
# 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"
     )
```

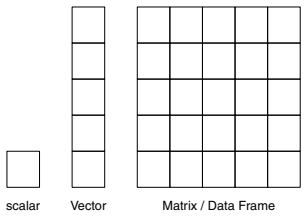


Figure 15: scalar, Vector, Matrix... ::drops mike::

cbind() and rbind()

`cbind()` and `rbind()` both create matrices by combining several vectors together into a single matrix. cbind() combines vectors as columns in the matrix, while rbind() combines them as rows.

cbind(), rbind()

`x, y, ...`

One or more vectors to be combined into a matrix

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 <- 1:10
y <- 11:20
z <- 21:30

matrix.1 <- rbind(x, y, z)
matrix.1

##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## x    1    2    3    4    5    6    7    8    9    10
## y   11   12   13   14   15   16   17   18   19    20
## z   21   22   23   24   25   26   27   28   29    30

matrix.2 <- cbind(x, y, z)
matrix.2

##        x  y  z
##  [1,]  1 11 21
##  [2,]  2 12 22
##  [3,]  3 13 23
##  [4,]  4 14 24
##  [5,]  5 15 25
##  [6,]  6 16 26
##  [7,]  7 17 27
##  [8,]  8 18 28
##  [9,]  9 19 29
## [10,] 10 20 30
```

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

If you want to create a matrix from a single vector of data, you can do this using the `matrix()` function.

`matrix()`

`data`

A vector of data

`nrow`

The number of rows in the final matrix

`ncol`

The number of columns in the final matrix

`byrow`

A logical value indicating whether to fill the matrix by row or column

Let's use the `matrix()` function to re-create a matrix containing the values from 1 to 30.

```
matrix.1 <- matrix(data = 1:30,
                   nrow = 10,
                   ncol = 3)
matrix.1
```

```
##       [,1] [,2] [,3]
##  [1,]    1   11   21
##  [2,]    2   12   22
##  [3,]    3   13   23
##  [4,]    4   14   24
##  [5,]    5   15   25
##  [6,]    6   16   26
##  [7,]    7   17   27
##  [8,]    8   18   28
##  [9,]    9   19   29
## [10,]   10   20   30
```

```
matrix.2 <- matrix(data = 1:30,
                   nrow = 3,
                   ncol = 10)
matrix.2
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    4    7   10   13   16   19   22   25    28
## [2,]    2    5    8   11   14   17   20   23   26    29
## [3,]    3    6    9   12   15   18   21   24   27    30
```

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"
```

Dataframe: An m x n object containing numbers, strings and factors

A dataframe looks a lot like a matrix at first: it is also rectangular and has m rows and n columns. However, unlike matrices, dataframes can contain *both* string vectors and numeric vectors within the same object. For this reason, most large datasets in R, for example, a survey including numeric data and text data, will be stored as dataframes.

A dataframe is just a more flexible matrix that allows you to combine both character and numeric vectors into the same data object. Because dataframes are more flexible than matrices, Most datafiles you use will be stored as dataframes.

`data.frame()`

To create a dataframe, you can use the `data.frame()` function.To use the `data.frame()` function, start by defining the name of a column as a string (for example, `"gender"`), then indicate the contents of the column as a vector. You can add as many columns as you would like.

Let's create a dataframe of fictional survey data. I'll add a column called `initials` with a string vector indicating the person's first and last initials. I'll then add a `gender` column with 5 string values. I'll then add a column called `"height"` with five height values, and a column called `"siblings"` with 5 integer values indicating how many siblings each person has. Finally, I'll add the argument `stringsAsFactors = F`. This tells R to interpret strings as strings, and not as Factors, which are a different datatype that we don't want to deal with just yet.

```
survey <- data.frame(
          "initials" = c("MP", "HT", "RH", "AP", "NP"),
           "gender" = c("F", "F", "F", "M", "M"),
           "height" = c(132, 150, 167, 148, 172),
           "siblings" = c(0, 2, 0, 1, 2),
           stringsAsFactors = F # don't convert strings to factors
          )

survey # Print the dataframe

##   initials gender height siblings
## 1       MP      F    132        0
## 2       HT      F    150        2
## 3       RH      F    167        0
## 4       AP      M    148        1
## 5       NP      M    172        2
```

Data sets pre-loaded in R

Until now, we've used the functions `matrix()` and `dataframe()` to manually create our own datasets within R. However, for demonstration purposes, it's frequently easier to use existing datasets. Thankfully, R has us covered: R has several datasets that come pre-installed in a package called `datasets`. 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 by by name. We'll use them in the following examples.

Getting information about matrices and dataframes

When you are working with dataframes, you will frequently want to know its general attributes, such as the number of rows and columns it has. Here are some common functions to get basic information about a dataframe. For each of these functions, I'll use the label df to refer to the dataframe object:

Helpful Dataframe Functions

`head(df), tail(df)`

Look at the first few rows (head()) or the last few rows (tail()) of a dataframe in the console.

`View(df)`

View the entire dataframe in a spreadsheet-like object. Make sure to use a capital V or the function won't work!

`dim(df), nrow(df), ncol(df)`

Return information about the dimensions of the dataframe. dim() returns the number of rows and columns, while nrow() and ncol() return just the number of rows or columns.

`names(df)`

Returns a string vector showing the names in the dataframe.

`summary(df)`

Returns summary information about all columns in the dataframe. For numeric columns, you'll see basic summary statistics like the minimum, mean, maximum etc.

`str(df)`

Tells you the structure of the dataframe - with information about the column names, the data class of each column (e.g.; numeric or string), and the first few cases in each column

Let's use some of these functions on the `ChickWeight` dataframe:

names(df)

The `names()` function will tell you the names of the columns in a dataframe. Let's use the function to see the columns in `ChickWeight`:

```
names(ChickWeight)

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

head(df)

The function `head(x)` will show you the first few rows of a matrix / dataframe. Personally, I am constantly using this function to

make sure that I didn't screw up a dataset when I'm working on it. Let's look at the first few rows of the dataframe `ChickWeight`, which contains data on the growth of chickens on several different diets.

```
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
```

The `head()` function only shows you the first few rows of a dataframe, but usually this is enough to get a visual sense of the names of the dataframe, the number of columns, and the type of data in each column. But what if you want to see all values? You could print the entire dataframe into the console, but the console isn't a very friendly environment to view data. Instead, you can use the `View()` function, which will print the entire dataframe into a spreadsheet-like window:

```
View(df)
```

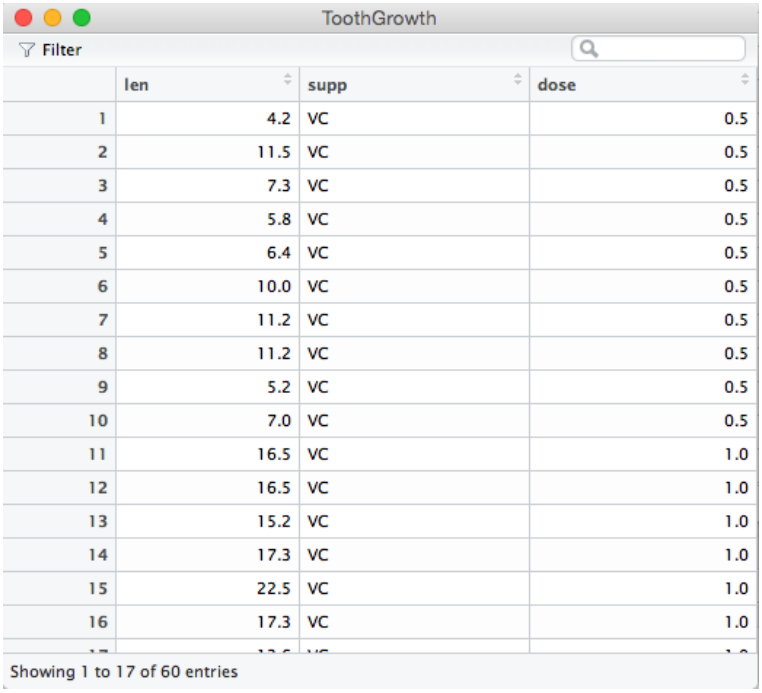
Let's use the `View()` function to look at the entire `ToothGrowth` dataframe:

```
View(ToothGrowth)
```

When you run this code, you should see a separate window open (see Figure 16). You can use this window to scroll through the data, sort it via column values (by clicking on the column name), and even apply filters using the filter button on the top left of the screen. However, keep in mind that anything you do in the `View()` window will *not* change the actual dataframe in any way. You cannot add or remove data using the window, and any sorting or filtering you apply won't be replicated in the actual data.

```
dim(df), nrow(df), ncol(df)
```

The `dim()`, `nrow()` and `ncol()` functions will give you information about the dimensions of the dataframe:



ToothGrowth

Filter

	len	supp	dose
1	4.2	VC	0.5
2	11.5	VC	0.5
3	7.3	VC	0.5
4	5.8	VC	0.5
5	6.4	VC	0.5
6	10.0	VC	0.5
7	11.2	VC	0.5
8	11.2	VC	0.5
9	5.2	VC	0.5
10	7.0	VC	0.5
11	16.5	VC	1.0
12	16.5	VC	1.0
13	15.2	VC	1.0
14	17.3	VC	1.0
15	22.5	VC	1.0
16	17.3	VC	1.0

Showing 1 to 17 of 60 entries

Figure 16: Screenshot of the window from `View(ToothGrowth)`. 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.

```
dim(ChickWeight)

## [1] 578   4

nrow(ChickWeight)

## [1] 578

ncol(ChickWeight)

## [1] 4
```

Loading data into R with read.table()

So far we've used either randomly generated data, or datasets pre-loaded in R. But how do you get an existing dataset into R? For the most part, getting datasets into R isn't that tricky - but only if your data is already in a 'nice' format. By 'nice,' I mean a text file with tab (or comma) separated columns. If your data is in another format (like Excel or Shitty Piece of Shitty Shit), I strongly recommend first exporting the data to a tab-delimited text file, and only then loading the data into R. That said, if for some reason you absolutely have to load a non-text file into R, look at the *Additional Tips* sections for instructions.

Import data into R as comma or tab-delimited text files whenever possible. If you need to load data in another format (e.g.; Excel), save it as a text file from the original program first.

Once you have a text file, you can load it into R using the `read.table()` function. To use the `read.table()` function, you need to know where the text file is located on your computer. To do this, find the file on your harddrive then right-click it and view its properties. You should be able to see its file-path there. For example, the file path of a text file called `mydata` on my desktop is `"Users/Nathaniel/Desktop/mydata.txt"`.

Here are the main arguments to `read.table()` (to see all of them, run `?read.table`)

`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 or not.

`ncol`

The number of columns in the final matrix

`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)

To test this function, let's read in the datafile called `pirates.txt`. This datafile contains data from a survey of 1,000 pirates at the 2015 annual pirate meeting at the Bodensee in Konstanz, Germany. You can access this data in one of two ways: First, you can download this file from: http://nathanieldphillips.com/wp-content/uploads/2015/11/pirates1.txt) and a note of its directory on your computer (on my computer, the path is `/Users/Nathaniel/Dropbox/Public/pirates.txt`). You can then load the data into R by using `read.table`:

```
pirate.survey <- read.table(file = "/Users/Nathaniel/Dropbox/Public/pirates.txt",
                      header = T,
                      sep = "\t", # tab-delimited
                      stringsAsFactors = F
                      )
```

Loading a dataset from a text file.

If you receive an error, it's probably because you entered the file path incorrectly. One trick to get the file path easily is by using RStudio's **Import Dataset** menu (see *Additional Tips*). If you got the directory location correct, and the file exists, then you should not receive any error warning after executing `read.table()`.

Alternatively, you can load the dataset directly into R by entering the HTML link as the `file` argument to `read.table`

```
pirate.survey <- read.table(file = "http://nathanieldphillips.com/wp-content/uploads/2015/11/pirates1.txt",
                      header = T,
                      sep = "\t", # tab-delimited
                      stringsAsFactors = F
                      )
```

Loading a data set from a URL link

The data is now stored as a dataframe and you can now access it via the object name you assigned it to (in my case, I called it `Flights`). To make sure it loaded correctly, try seeing the first few rows with `head()`

```
head(pirate.survey)

##   id    sex headband age college tattoos tchests parrots favorite.pirate
## 1  1 female      yes  30   JSSFP      11      20       7      Blackbeard
## 2  2   male      yes  25    CCCC      15       6       3        Anicetus
## 3  3   male      yes  25    CCCC      12       5       2    Jack Sparrow
## 4  4   male      yes  29   JSSFP      12       0       1      Edward Low
## 5  5 female      yes  31    CCCC      17      11      10        Anicetus
## 6  6   male      yes  30   JSSFP      12       2       2    Jack Sparrow
##   sword.type sword.time eyepatch beard.length       fav.pixar
## 1    cutlass       0.83        1            0              Up
## 2    cutlass       0.03        1           16     Toy Story 2
## 3    cutlass       1.44        0           21            Cars
## 4    cutlass       0.18        1           21 The Incredibles
## 5    cutlass       0.64        1            0      Inside Out
## 6      sabre      13.90        1           22      Inside Out
```

Additional tips

- If you're like me, and you hate figuring out (and typing) the directory of a file, you can use RStudio's menu to help you. If you

click on the Environment window and click the button `Import Dataset`, you'll activate a menu that will allow you to select the file using your computer's finder. You'll then be greeted with a graphical interface for setting the import parameters. When you are finished, RStudio not only import the dataset, but it will paste the R code needed to import the data into the console. You can then copy the code (which includes the file path) and paste it into your R document so the next time you use the document you can just run the code to import the data.

- 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!

Here are four vectors with information about 10 pirates

```
id <- 1:10
sex <- c("female", "male", "male", "male", "female", "male", "female",
"female", "male", "female")
age <- c(30, 25, 25, 29, 31, 30, 33, 35, 25, 34)
tattoos <- c(11, 15, 12, 12, 17, 12, 9, 13, 9, 9)
```

1. Combine the vectors id, sex, age, and tattoos above into a single dataframe called `pirates.sample`
2. What is the median age of the 10 pirates?

   For the next questions, we'll use the `movies` dataset in the yarrr package.
3. How many rows and columns are in the movies dataset? What are the column names?
4. Show me all the data for Harry Potter and the Chamber of Secrets
5. What was the boxoffice total of Harry Potter and the Deathly Hallows?

6. What percent of the movies were sequels? (Hint: Use logical indexing and mean())

7. How many movies were there of each genre?

8. What were the names of the movies that were made before 1950? What was the mean running time of those movies in minutes?

9. Create a new column called revenue.d.budget which shows a movieâĂŹs boxoffice total divided by its budget. Which movie had the highest box office total relative to its budget? Which comedy movie had the highest box office total relative to its budget?

10. How many movies made less than $30 Million dollars AND were Comedies? (hint: Use subset() and nrow()). What were the names of these movies?

6: Basic Dataframe Manipulation

Chapter Goals

1. Indexing dataframes with brackets [,], and $
2. Subsettting dataframes with logical indexing and subset()
3. Recoding values in a dataframe with indexing

In this chapter we'll cover how to do some basic analyses on dataframes. We'll focus on dataframes, and not on matrices, because most datasets you use will be stored as dataframes. However, if you do find yourself working with matrices, many of the techniques you'll learn in this chapter will also apply to them.

Get the pirates dataframe

First thing's first - let's get access to the pirates dataframe. If you've already installed and loaded the yarrr package (see Chapter 1), you should already have access to it. If you are unable to install the yarrr package, you can also download the dataset directly from the web using the following code:

```
pirates <- read.table(file = "http://nathanieldphillips.com/wp-content/uploads/2015/11/pirates1.txt",
                      header = T,
                      sep = "\t", # tab-delimited
                      stringsAsFactors = F
                      )
```

To make sure it worked, try running the following code which should show you the first few rows of the dataset

```
head(pirates)
```

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 scalars or vectors of the row and column numbers you want to get.

Let's try this on the `pirates` dataframe. First, let's look at the entry in row 1, and column 1. Since the first column is the pirate's id, this will be the id of the first pirate (it should be 1):

```
pirates[1, 1]

## [1] 1
```

Now, let's look at the first 5 rows in columns 8 through 10:

```
pirates[1:5, 8:10]

##   parrots favorite.pirate sword.type
## 1       7      Blackbeard    cutlass
## 2       3        Anicetus    cutlass
## 3       2    Jack Sparrow    cutlass
## 4       1      Edward Low    cutlass
## 5      10        Anicetus    cutlass
```

If you want to look at an entire row or an entire column, you can leave that index blank. For example, to see the entire first row of the pirates dataframe, we can leave the column index blank (but make sure to still use a comma):

```
pirates[1,]

##   id    sex headband age college tattoos tchests parrots favorite.pirate
## 1  1 female      yes  30   JSSFP      11      20       7      Blackbeard
##   sword.type sword.time eyepatch beard.length fav.pixar
## 1    cutlass       0.83        1            0        Up
```

You can use the same logic to get an entire column of a dataframe by leaving the index for rows blank. If you leave both index values blank, you'll get the entire dataframe back (which is the same thing as not using any indexing at all).

Of course, you can use any vector index you'd like (as long as the values are integers). For example, if I wanted every 100th entry in the 14th column (this column is the pirate's favorite pixar movie), I could set the row index to be the sequence of numbers from 1 to the number of rows in the dataframe in steps of 100.

```
row.index <- seq(from = 1, to = nrow(pirates), by = 100) # Create the row index
pirates[row.index, 14] # Give me every 100th entry in the 3rd column

##  [1] "Up"             "Inside Out"     "Up"             "Monsters, Inc."
##  [5] "Inside Out"     "Finding Nemo"   "Toy Story"      "Inside Out"
##  [9] "Toy Story"      "Finding Nemo"
```

Here you can see the benefits of using `nrow()` - I used it to make sure I gave valid index values to `pirates`

Accessing dataframe columns by column name and $

One of the nice things about dataframes is that each column will have a name. You can then use this name to access specific columns without having to index columns by numbers. 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 `pirates` dataframe:

```
names(pirates)

##  [1] "id"              "sex"             "headband"
##  [4] "age"             "college"         "tattoos"
##  [7] "tchests"         "parrots"         "favorite.pirate"
## [10] "sword.type"      "sword.time"      "eyepatch"
## [13] "beard.length"    "fav.pixar"
```

To access a specific column in a dataframe by name, you use the the `$` operator:

```
dataframe$colname
```

where `dataframe` is the name of the dataframe, and `colname` is the name of the column you are interested in. When you apply the `$` operator to a dataframe, it will return a vector. Let's access some of the vectors in the dataframe `pirates`:

```
pirates$age
pirates$sex
```

The commands above will print the two columns into the console.

Because the $ operator returns a vector, you can easily calculate descriptive statistics on columns of a dataframe using $. Let's calculate the mean age of all pirates, and then create a table showing how many pirates there are of each sex:

```
mean(pirates$age)

## [1] 27.395

table(pirates$sex)

##
## female   male  other
##    491    480     29
```

It looks like the average age of the pirates is 27.39. There are also 491 female pirates, 480 pirates, and 29 pirates who indicated that they were neither male nor female.

Adding new columns to a dataframe

You can easily add 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 `pirates` called `tattoos.per.year` which indicates the number of tattoos a pirate has divided by his/her age:

```
pirates$tatoos.per.year <- pirates$tattoos / pirates$age
```

You can add new columns with any information that you want to a dataframe - even basic numerical vectors. For example, let's say that I needed to assign all 1,000 to work detail on one of two ships: the Kantine and the Blechnerei. I'll add a new column to the dataframe called "assigned.ship" which is a vector repeating the strings "Kantine" and "Blechnerei" over and over 1,000 times

```
pirates$assigned.ship <- rep(c("Kantine", "Blechnerei"), length.out = 1000)
```

Warning: Always repeat the name of the dataframe when referring to it!

When you are conducting analyses on dataframes, it's important that you always repeat the name of the dataframe when accessing its columns. If you don't, R will assume the column name is a totally different object.

For example, let's say I wanted to add a column to the pirates dataframe called "tchests.per.year" that shows how many treasure chests the pirate found on average in every year of his/her life. To do this, I simply need to divide the tchests column by the age column.

Now, the following code *won't* work because we didn't specify the pirates dataframe before each column name

```
# This won't work because we didn't specify the pirates dataframe !!
tchests.per.year <- tchests / age
```

Here is the proper code where we specify the pirates dataframe and the *operatorbeforeeachcolumnname* :

```
pirates$tchests.per.year <- pirates$tchests / pirates$age
```

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, the name of the first column in the `pirates` dataset is `id`. Let's change it to `pirate.id`.

```
names(pirates)[1] <- "pirate.id"
```

Now let's look at the new list of names to make sure it worked. The first name should now be `pirate.id`

```
names(pirates)
```

```
##  [1] "pirate.id"       "sex"             "headband"
##  [4] "age"             "college"         "tattoos"
##  [7] "tchests"         "parrots"         "favorite.pirate"
## [10] "sword.type"      "sword.time"      "eyepatch"
## [13] "beard.length"    "fav.pixar"       "tatoos.per.year"
## [16] "assigned.ship"
```

To use indexing like we did above, I had to know that the `id` column was the first column in the dataframe. But what if you don't know exactly where the name occurs? For example, let's say we want to change the column `college` to `pirate.college`, but we don't know the index of the name. We can do this by adding logical indexing to the names vector. First, let's figure out where the `college` column is:

```
log.vec <- names(pirates) == "college"
which(log.vec)
```

```
## [1] 5
```

It looks like the `college` column is the 5 column. Now let's update the name using `log.vec` as the index to `names`

```
names(pirates)[log.vec] <- "pirate.college"
names(pirates) # Make sure it worked

##  [1] "pirate.id"       "sex"             "headband"
##  [4] "age"             "pirate.college"  "tattoos"
##  [7] "tchests"         "parrots"         "favorite.pirate"
## [10] "sword.type"      "sword.time"      "eyepatch"
## [13] "beard.length"    "fav.pixar"       "tatoos.per.year"
## [16] "assigned.ship"
```

Now that we're done with the example, I'm going to change the names back to their original values

```
names(pirates)[names(pirates) == "pirate.id"] <- "id"
names(pirates)[names(pirates) == "pirate.college"] <- "college"
```

It can get annoying sometimes to constantly have to type the name of the dataframe every time you are referring to it. To help you reduce typing, there is a function `with()` that can help prevent you from having to repeat the name of a dataframe over and over again.

with(x, ...)

`with(x, ...)`: Simplifies your code for dataframe manipulation by allowing you to just enter the name of the dataframe once.

The function `with()` allows you to specify a dataframe (or any other object in R) once. Then, for every object you refer to in the code in that line, R will assume you're referring to that object in an expression.

For example, let's repeat the `pirates$tchests.per.year` calculation using `with()`. We'll set the name of the dataframe as the first argument, then do our regular calculations on the column names.

```
pirates$tchests.per.year <- with(pirates, tchests / age)
```

Subsetting dataframes with logical indexing and subset()

Many, if not all, of the analyses you will be doing will be on subsets of data, rather than entire datasets. For example, we might want to calculate the mean age of female pirates separately from male pirates. Or, we might want to calculate the median beard length of pirates who wear headbands and compare that to the median length of those who don't wear headbands (hey why not??)

Subsetting - selecting subsets of data based on some criteria - is very easy in R. To do this, we can use one of two methods: indexing with logical vectors, or the `subset()` function. We'll start with indexing first.

Indexing dataframes with logical vectors is very similar to indexing data vectors. First, we create a logical vector. Next, we index the dataframe using that logical vector.

Let's use indexing to access just the data for pirates who never had a parrot in `pirates`. We'll assign this subset of the data to a new dataframe called pirates.s1 indicating that it's our first subset (s1) of the pirates data:

```
  # Step 1: Create a logical vector
noparrots.log <- pirates$parrots == 0
  # Step 2: Index dataframe by logical vector
pirates.s1 <- pirates[noparrots.log ,]
```

If you'd like, you can also combine the two steps in one line. For example, the following code gives the same result as the previous:

```
  # Two steps in 1
pirates.s1 <- pirates[pirates$parrots == 0, ]
```

Now, let's try indexing the `pirates` data using a slightly more complicated index. For example, let's access just the data for pirates where age is less than 25 *and* the college is `"CCCC"`. We'll assign this data to a new dataframe called pirates.s2

```
pirates.s2 <- pirates[pirates$age < 20 & pirates$college == "CCCC",]
nrow(pirates.s2)

## [1] 79
```

Looks like we have 79 pirates who are younger than 25 and who went to Captain Chunk's Canon Crew.

Indexing with brackets is the standard way to slice and dice dataframes. However, if you are working on data that is all in the same dataframe, it can get a bit tiresome to have to constantly repeat the name of the dataframe. For example, let's say we wanted to get data from `pirates` where `age < 30` and `college == "CCCC"` and `tchests.found >= 35`. We could do this with indexing but it would take a lot of code. A way to get around having to repeat the name of the dataframe over and over is to use the `subset()` function.

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

Let's use the subset() command to create a new dataframe called pirates.s3 which contains the data of pirates who are female (sex == "female") and who use a cutlass sword (sword.type == "cutlass") and who have less than 10 tattoos (tattoos < 10). We'll call this new object pirates.s3

```
pirates.s3 <- subset(x = pirates,
                subset = (sex == "female" &
                          sword.type == "cutlass" &
                          tattoos < 10)
                )
```

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. For example, let's say I just want the `id` and `age` columns from the previous analysis. To do this, I'll just add the column names as inputs to the select argument:

```
pirates.s4 <- subset(x = pirates,
                subset = (age < 30 &
                          college == "CCCC" &
                          tchests >= 35),
                select = c("id", "age"))

head(pirates.s4)
```

```
##      id age
## 120 120  27
## 255 255  25
## 415 415  23
## 428 428  29
```

```
## 526 526  27
## 856 856  20
```

Combining indexing and functions

Once you know how to index a dataframe to get the data vectors you want, you can then easily apply functions like descriptive statistics based on specific criteria. For example, let's calculate the mean age of the pirates who went to Captain Chunk's Canon Crew (`college == "CCCC"`). To show you that there are many ways to do this, I'll write the code in three different ways.

First, let's use logical indexing

```
mean(pirates$age[pirates$college == "CCCC"])

## [1] 24.48126

# OR using the with() function
with(pirates, mean(age[college == "CCCC"]))

## [1] 24.48126
```

Now let's do it using the subset() command. First we'll create a new dataframe called pirates.s5 only containing data for pirates that went to Captain Chunk's Cannon Crew (`college == "CCCC"`). Then we'll calculate the mean of the age column in the new dataframe:

```
pirates.s5 <- subset(pirates,
                     subset = college == "CCCC")

mean(pirates.s5$age)

## [1] 24.48126
```

In fact, we can make the previous code even simpler by combining the `with()` function and the `subset()` function:

```
with(subset(pirates, college == "CCCC"),
     mean(age)
     )

## [1] 24.48126
```

As you can see, there are many ways to do the same thing in R. Ultimately, the choice of which specific code and functions you use is up to you.

Recoding values in a dataframe column

Let's say you have a dataframe with some messed up values - for example, a survey where someone gave an invalid response. How can you convert the messed up values to reasonable ones? We've done this before with vectors, and it's just as easy with columns in a dataframe. Just create a logical vector indicating which values you want to change, and reassign them to the values you want.

Let's start by creating a copy of the pirates dataframe (just so we don't permanently mess up the original). We'll call this dataframe `pirates.copy`:

```
pirates.copy <- pirates
```

Now, let's look at the responses to the column `sword.type`

```
table(pirates.copy$sword.type)

##
##   banana  cutlass    sabre scimitar
##       44      830       60       66
```

As you can see, 44 pirates said they used a banana as a sword. Assuming this is an invalid response, we might want to recode all the "banana" responses to NA.

```
log.vec <- pirates.copy$sword.type == "banana"
pirates.copy$sword.type[log.vec] <- NA
```

Now let's check the table of responses again

```
table(pirates.copy$sword.type)

##
##  cutlass    sabre scimitar
##      830       60       66
```

As you can see, all the "banana" responses are gone!

If you want to include totals of NA values in the `table()` function, you need to include the argument `useNA = "always"`

```
vec <- c(1, 1, 1, NA, NA, NA)

table(vec)

## vec
## 1
## 3

# Now include NA totals
table(vec, useNA = "always")

## vec
##    1 <NA>
##    3    3
```

A worked example: movies

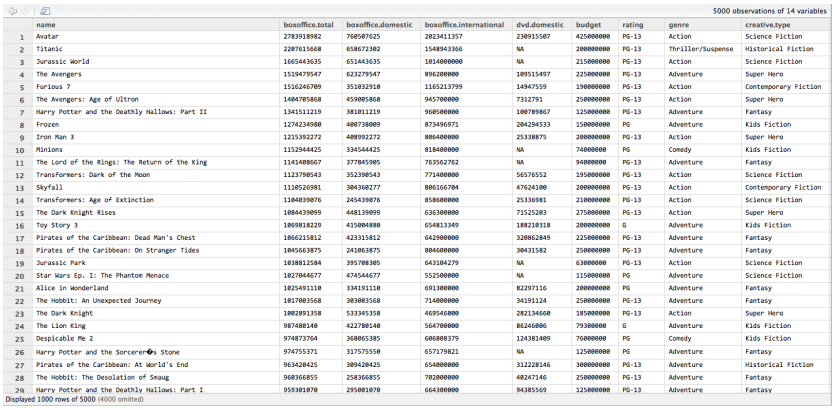
For this example, we'll work with a new dataset called `movies`. This dataset contains information about the top 5,000 grossing movies of all time.If you've downloaded the yarrr package, the dataset is already on your computer (you can look at documentation for the data by running `?movies`. If you don't have the yarrr package, you

can download the data and assign it to an object called `movies` using the following code:

```
movies <- read.table("http://nathanieldphillips.com/wp-content/uploads/2015/11/movies.txt",
                     sep = "\t",
                     header = T,
                     stringsAsFactors = F)
```

Here is a screenshot of the dataset:

```
View(movies)
```



5000 observations of 14 variables

	name	boxoffice.total	boxoffice.domestic	boxoffice.international	dvd.domestic	budget	rating	genre	creative.type
1	Avatar	2783918982	760507625	2023411357	230915507	425000000	PG-13	Action	Science Fiction
2	Titanic	2207615668	658672302	1548943366	NA	200000000	PG-13	Thriller/Suspense	Historical Fiction
3	Jurassic World	1665443635	651443635	1014000000	NA	215000000	PG-13	Action	Science Fiction
4	The Avengers	1519479547	623279547	896200000	109515497	225000000	PG-13	Adventure	Super Hero
5	Furious 7	1516246709	351032918	1165213799	14947559	190000000	PG-13	Action	Contemporary Fiction
6	The Avengers: Age of Ultron	1404705868	459005868	945700000	7312791	250000000	PG-13	Action	Super Hero
7	Harry Potter and the Deathly Hallows: Part II	1341511219	381011219	960500000	100789867	125000000	PG-13	Adventure	Fantasy
8	Frozen	1274234980	400738009	873496971	204294533	150000000	PG	Adventure	Kids Fiction
9	Iron Man 3	1215392272	408992272	806400000	25338875	200000000	PG-13	Action	Super Hero
10	Minions	1152944425	334544425	818400000	NA	74000000	PG	Comedy	Kids Fiction
11	The Lord of the Rings: The Return of the King	1141408667	377845905	763562762	NA	94000000	PG-13	Adventure	Fantasy
12	Transformers: Dark of the Moon	1123790543	352390543	771400000	56576552	195000000	PG-13	Action	Science Fiction
13	Skyfall	1110526981	304360277	806166704	47624100	200000000	PG-13	Action	Contemporary Fiction
14	Transformers: Age of Extinction	1104039076	245439076	858600000	25336981	210000000	PG-13	Action	Science Fiction
15	The Dark Knight Rises	1084439099	448139099	636300000	71525203	275000000	PG-13	Action	Super Hero
16	Toy Story 3	1063918229	415004880	654813349	18821031	200000000	G	Adventure	Kids Fiction
17	Pirates of the Caribbean: Dead Man's Chest	1066215812	423315812	642900000	32086284	225000000	PG-13	Adventure	Fantasy
18	Pirates of the Caribbean: On Stranger Tides	1045663875	241063875	804600000	30431582	250000000	PG-13	Adventure	Fantasy
19	Jurassic Park	1038812584	395708305	643104279	NA	63000000	PG-13	Action	Science Fiction
20	Star Wars Ep. I: The Phantom Menace	1027044677	474544677	552500000	NA	115000000	PG	Adventure	Science Fiction
21	Alice in Wonderland	1025491110	334191110	691300000	82297116	200000000	PG	Adventure	Fantasy
22	The Hobbit: An Unexpected Journey	1017003568	303003568	714000000	34191124	250000000	PG-13	Adventure	Fantasy
23	The Dark Knight	1002891358	533345358	469546000	28213466	185000000	PG-13	Action	Super Hero
24	The Lion King	987480140	422780140	564700000	86246006	79300000	G	Adventure	Kids Fiction
25	Despicable Me 2	974873764	368065385	606808379	12430140	76000000	PG	Comedy	Kids Fiction
26	Harry Potter and the Sorcerer<U+FFFD>s Stone	974755371	317575550	657179821	NA	125000000	PG	Adventure	Fantasy
27	Pirates of the Caribbean: At World's End	963420425	309420425	654000000	31222814	300000000	PG-13	Adventure	Historical Fiction
28	The Hobbit: The Desolation of Smaug	960366855	258366855	702000000	40247146	250000000	PG-13	Adventure	Fantasy
29	Harry Potter and the Deathly Hallows: Part I	955301070	295001070	664300000	94385569	125000000	PG-13	Adventure	Fantasy

Displayed 1000 rows of 5000 (4000 omitted)

Figure 17: Screenshot of the movies dataset.

Let's start by looking at the names of the dataset using the `names()` function:

```
names(movies)

##  [1] "name"                          "boxoffice.total"
##  [3] "boxoffice.domestic"            "boxoffice.international"
##  [5] "dvd.domestic"                  "budget"
##  [7] "rating"                        "genre"
##  [9] "creative.type"                 "time"
## [11] "year"                          "production.method"
## [13] "sequel"                        "boxoffice.domestic.inflationadj"
## [15] "budget.millions"               "boxoffice.total.millions"
## [17] "revenue.d.budget"
```

Let's answer 5 questions with this dataset:

Q1: What was the mean box-office total of Action movies and Adventure movies separately?

In this example (and in future examples), I'm combining the `with()` and `subset()` functions. This means that I'm first telling R to use the particular subset of the data that I want, and only then do the main calculation (like the `mean`).

```
action.mean <- with(subset(movies, genre == "Action"),
                    mean(boxoffice.total, na.rm = T))

action.mean # Print result

## [1] 148745191

adventure.mean <- with(subset(movies, genre == "Adventure"),
                    mean(boxoffice.total, na.rm = T))

adventure.mean # Print result

## [1] 234682919
```

Answer: The mean box office total of action movies was 149 Million. For adventure movies, the mean was 235 Million.

Q2: Was there a difference in the production budget of Super Hero versus Science Fiction movies?

```
# First let's convert the budget to millions
movies$budget.millions <- movies$budget / 1000000

sh.budget.summary <- with(subset(movies, creative.type == "Super Hero"),
     summary(budget.millions))
sh.budget.summary

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0    60.0   120.0   117.0   152.5   275.0

sf.budget.summary <- with(subset(movies, creative.type == "Science Fiction"),
     summary(budget.millions))
sf.budget.summary

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00   10.00   38.00   55.71   80.00  425.00
```

Answer: Super Hero movies had higher budgets on average than Science Fiction movies. For example, the median budget of Super Hero movies was 120 Million, while median budget of Science Fiction movies was just 38.

Q3: Were are higher percentage of R rated movies in the Action genre than the Thriller/Suspense genre?

Any time you need to answer a percentage question, it's almost always a good idea to first convert the data to a logical vector, where target cases are TRUE, and then take the mean. In this example, I'm calculating the mean of the logical vector `rating == "R"`.

```
action.r.percentage <- with(subset(movies, genre == "Action"),
                            mean(rating == "R", na.rm = T))
action.r.percentage

## [1] 0.4894737

thriller.r.percentage <- with(subset(movies, genre == "Thriller/Suspense"),
                              mean(rating == "R", na.rm = T))
thriller.r.percentage

## [1] 0.6216216
```

Answer: Of the action movies, 49% were rated R. Of the Thriller / Suspense movies, the percentage was 62%

Q4: Did movies with a running time longer than 100 minutes make more money than those with a running time less than 100 minutes??

```
# First let's create a new column giving boxoffice total in
#  millions

movies$boxoffice.total.millions <- movies$boxoffice.total / 1000000

# Now we'll calculate the averages

earnings.gt100 <- with(subset(movies, time > 100),
                       mean(boxoffice.total.millions))
earnings.gt100

## [1] 180.5893

earnings.lt100 <- with(subset(movies, time < 100),
                       mean(boxoffice.total.millions))
earnings.lt100

## [1] 125.5432
```

Answer: Longer movies tended to make more money. Movies with a running time greater than 100 minutes made 180.59 Million on average, while those with a running time less than 100 made 125.54 Million on average.

Q5: What was the name of the movie that earned the highest box-office relative to its production budget?

```
movies$revenue.d.budget <- with(movies, boxoffice.total.millions / budget.millions)
max.val <- with(subset(movies, budget.millions > 0), max(revenue.d.budget))
max.val # Print the maximum percentage increase (not necessary)

## [1] 4138.333

movies$name[movies$revenue.d.budget == max.val]

## [1] "The Blair Witch Project"
```

Question 5 is a bit tricky. First, I calculated a new column that divided a movie's box-office revenue by it budget called `revenue.d.budget`. Next, I calculated the maximum `revenue.d.budget` value in the dataset but *only* for those movies with a budget greater than 0. Finally, I indexed the names of the movies by only looking at the rows where the `revenue.d.budget` was equal to the maximum `revenue.d.budget` value.

Answer: The Blair Witch Project made 4138 times its production budget!

Test your R Might!

1. What was the box-office total of `"Life of Pi"`? What was its budget?
2. What was the median budget of movies made in the year 2009? What percent of these movies were sequels?
3. What percent of Science Fiction movies (look in the `creative.type` column) were rated PG-13? Of these movies, what was the median budget?
4. How many Action movies had a budget less than 150 million? Of these movies, what was the name of the movie with the highest box-office total and how much did it earn?
5. Which movie had the worst box-office total relative to its budget? What was the genre of this movie?

7: Grouped aggregation with dataframes

Chapter Goals

1. Grouped aggregation with aggregate() and dplyr

Grouped aggregation with aggregate()

Many of the questions we might want to answer with a dataset have to do with comparisons between groups. For example, in our pirates dataset we could ask "What is the average age of pirates from each college?" or "Do older pirates tend to have faster sword speeds?" In each of these questions, we want to know a descriptive statistic of a numeric variable (age and sword speed) as a function of one or more independent variables (college and age). By now, your R skills are good enough that you *could* answer these questions already. You could use `subset()` or logical indexing to slice and dice the data set for each level of the independent variable. However, it would be a pain to have to manually create new subsets or indexes for each level of the independent variable. Thankfully, R contains many functions that will help you do this in a snap.

The first 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, ...). 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 the function to answer the question "What is the average (mean) age of pirates from each college?" For this question, we'll set the value of the dependent variable Y to age, x1 to college, and FUN to mean

```
aggregate(formula = age ~ college,  # DV is cancelled, IV is carrier
          FUN = mean, # Calculate the mean of the DV for each IV level
          na.rm = T, # Ignore NA values when calculating the mean
          data = pirates # IV and DV are located in the Flights dataframe
          )
```

```
##   college      age
## 1    CCCC 24.48126
## 2   JSSFP 33.23123
```

As you can see, the `aggregate()` function has returned a dataframe with a column for the independent variable (`college`), and a column for the results of the function `mean` applied to each level of the independent variable. We can easily plot these data using the `barplot()` function, which plots a numeric variable as a function of a nominal variable (see margin Figure 18)

You can include any function as the argument to `FUN` as long as the function takes a single numeric argument and returns a single scalar. You can also include multiple independent variables. For example,

```
aggregated.data <- aggregate(formula = age ~ college,
                             FUN = mean, na.rm = T, data = pirates)

barplot(height = aggregated.data$age,
        names.arg = aggregated.data$college)
```

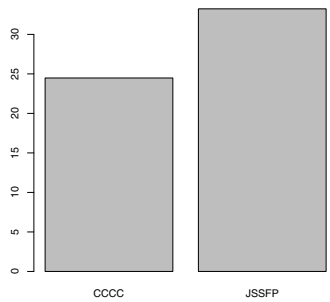


Figure 18: Barplot showing the mean age for each pirate college

let's use `aggregate()` to now get the maximum age (i.e.; the oldest pirate) for all combinations of college and headband:

```
# Calculate median departure delay by carrier
agg.data <- aggregate(
            formula = age ~ college + headband, # DV is age, IV is college and headband
            FUN = max, # Calculate the median age
            data = pirates, # Columns are in the pirates dataframe
            na.rm = T) # Ignore NA values when calculating the median

agg.data # print the result!

##   college headband age
## 1    CCCC       no  32
## 2   JSSFP       no  45
## 3    CCCC      yes  34
## 4   JSSFP      yes  43
```

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 (like age, sword speed, height, etc.), 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 dplyr

The `dplyr` package is a relatively new R package that allows you to do all kinds of analyses quickly and easily. 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.

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 original (raw) dataframe is called `my.df`, the variable you want to collapse the data over is 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 = mean(col.a),  # calculate mean of column col.a in my.df
      b.sd = sd(col.b),      # calculate sd of column col.b in my.df
      c.max = max(col.c)     # calculate max on column col.c in my.df, ...
      )
```

Here's how you should think about the code above:

Start with the dataframe `my.df`. *Then*, group `my.df` by the grouping variable `grouping.variable`. *Then*, calculate the following summary columns in the dataset: `a.mean` should be the mean of `col.a` in my.df, `b.sd` should be the standard deviation of `col.b` in my.df, and `c.max` should be the maximum value of `col.c` in my.df.

When you use dplyr, you write code that sounds like: "The original dataframe is XXX, now filter the dataframe to only include rows that satisfy the conditions YYY, now group the data at each level of the variable(s) ZZZ, now summarize the data and calculate summary functions XXX..."

Let's start with an example: Let's create a dataframe of aggregated data from the `pirates` dataset. I'm going to group the data according to the columns college and headband. where each row is a unique combination of college and headband. I'll then create several columns is a different summary statistic of some data across all pirates within each grouping of college and headband. Specifically, let's create 5 columns: `age.med`, The median age of pirates, `age.min`: The minimum age of pirates, `sword.time.med`: The median sword swing time of pirates, and `tchests.mean`: The average number of treasure chests found of pirates. Finally, I will calculate how many pirates were in each group using the n() function - a function specific to dplyr that counts cases.

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 `college.agg`:

```
library(dplyr)
college.agg <- pirates %>% # Define dataframe, THEN...
                group_by(college, headband) %>% # Define the grouping variable, THEN...
  summarise( # Tell R you are going to calculate summaries
    age.med = median(age), # Define first summary...
    age.min = min(age),  # Define second summary...
    sword.time.med = median(sword.time),
   tchests.mean = mean(tchests),
   n = n() # How many are in each group?
  ) # End

college.agg # Print result!
```

```
## Source: local data frame [4 x 7]
## Groups: college [?]
##
##   college headband age.med age.min sword.time.med tchests.mean     n
##     (chr)    (chr)   (dbl)   (dbl)          (dbl)        (dbl) (int)
## 1    CCCC       no      25      17           4.10     6.031250    64
## 2    CCCC      yes      25      12           0.58     6.875622   603
## 3   JSSFP       no      34      29           4.90     7.676471    34
## 4   JSSFP      yes      33      25           0.58     7.949833   299
```

As you can see, our final object `college.agg` is the aggregated dataframe we want which aggregates all the columns we wanted for each college and headband use. Let's walk through the code

When you use dplyr, the output will always be an object called a *local data frame*. A local dataframe is identical to a regular dataframe, except that it looks a bit nicer if you print the entire dataframe into the console. This means you don't have to use the `head()` function when looking at a local dataframe.

- First, we define the original dataframe that we are basing our summary statistics on. In this case, the original dataframe is `pirates`. We then include the pipe `%>%` to tell R we are still working.

- Second, we define the grouping variable using the `group_by` function. This tells R to group the results at the level of college and headband. We then use the `%>%` pipe.

- Next, we call the `summarise` function, which tells R that the following functions will be summaries of the grouping variable. Because all the arguments to the `summarise` function are within the parentheses, we don't need to use a pipe.

- Finally, we define the summary columns in our final dataframe. For each column, we give it a name (e.g.; `age.med`), and then write the calculation as a function of the appropriate columns in the original dataframe. For example, to define `age.med`, we write `median(age)`. The new function at the end is n(), which counts the number of cases in each group.

Hopefully you can see that this `dplyr` code is *much* simpler than the code we'd have to use if we wanted to create all these summary columns using `aggregate`.

The 5 verbs in dplyr

In the example above, we used the `dplyr` verb `summarise`. However, dplyr has other verbs that are just as useful:

dplyr verbs

`filter`

Select a subset of rows in a dataframe. For example, filter(sex == "male") will only include data where the column sex is male. Filter works identically to the subset() function in base-R.

`arrange`

Reorders rows of a dataframe according to a column. For example, arrange(age) will sort the dataframe according to an age column.

`select`

Select specific columns of a dataframe. There are many ways to use select, for some examples, check out http://www.r-bloggers.com/the-complete-catalog-of-argument-variations-of-select-in-dplyr/.

`mutate`

Add a column to a dataframe. For example, `mutate(bmi = weight / height)` will create a new column called bmi that divides a column weight by height

`summarise`

Creates summary columns as a function of columns in the original dataframe. Note: Only use after specifying `group_by` variables.

Let's do an example where we combine multiple verbs into one chunk of code. We'll create a new dataframe called `college.favpir.agg` that gives us aggregated data at the level of the both the college the pirate went to, and their favorite pirate. However, let's add some additional data filters this time. We'll filter the data to only include pirates who are older than 30 (`age > 40`)) and who wear a headband (`headband == "yes"`):

```
require(dplyr)
college.favpir.agg <- pirates %>% # First, define the original df
            filter(age > 30 & headband == "yes") %>% # Filter by age and headband
            group_by(college, favorite.pirate) %>% # Define the grouping variable
            summarise( #
              frequency = n(), # How many pirates in each group?
              sword.time.med = median(sword.time),  # Median sword speed?
              parrots.mean = mean(parrots) # # mean parrots?
            ) %>%
            arrange(frequency) # Step 7
```

```
college.favpir.agg # Print the result!

## Source: local data frame [12 x 5]
## Groups: college [2]
##
##    college favorite.pirate frequency sword.time.med parrots.mean
##      (chr)           (chr)     (int)          (dbl)        (dbl)
## 1     CCCC      Blackbeard         1          6.360     4.000000
## 2     CCCC      Lewis Scot         2          0.285     4.500000
## 3     CCCC        Anicetus         3          0.430     5.333333
## 4     CCCC      Edward Low         3          0.560     3.333333
## 5     CCCC            Hook         4          0.300     3.250000
## 6     CCCC    Jack Sparrow         8          0.350     4.500000
## 7    JSSFP      Blackbeard        23          0.590     3.086957
## 8    JSSFP        Anicetus        25          0.570     3.280000
## 9    JSSFP            Hook        31          0.580     3.419355
## 10   JSSFP      Lewis Scot        34          0.675     3.029412
## 11   JSSFP      Edward Low        47          0.550     4.340426
## 12   JSSFP    Jack Sparrow        72          0.510     3.041667
```

As you can see, our result is a dataframe with 12 rows and 5 columns. Let's walk through the code line by line:

1. First, we define the original dataframe as `pirates`, (%>% then...)
2. Next we filter the `pirates` dataframe by only including rows where age is greater than 30 and headband is yes (%>% then...)
3. We group the data according to `college` and `favorite.pirate` (%>% then...)
4. We call the `summarise` verb, telling dplyr that the next commands are summary functions of `pirates`. These will be the columns in our new aggregated dataframe. (%>% then...)
5. The first column in our new aggregated dataset is called `frequency` and is defined as the number of pirates in the group (the function `n()` is special to `dplyr` and simply returns the number of rows in a group)
6. The second column is called `sword.time.med` and is the median sword time of pirates in the group
7. The third column is called `parrots.mean` and is the mean number of parrots owned by pirates in the group.
8. After closing the `summary()` function, we arrange the final dataframe by the new column `frequency`

Test your R might!

1. Using the movies dataset, create a dataframe called `genre.agg` that shows the median budget of movies for each genre. Do this once using `aggregate()` function and once using dplyr.

2. Create a dataframe called `rating.agg` that groups the movies dataset by movie ratings and provides the following summary statistics: budget.mean, the mean movie budget – sequel.p, the proportion of movie sequels – and n, the number of movies with that rating.

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 have only scratched the surface of everything you can do with `dplyr`. For more `dplyr` tips, check out https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html

7: Sampling and Probability Distributions

Sampling data from probability distributions

```
# 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)
```

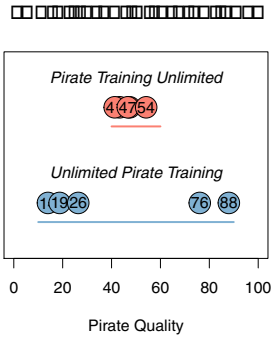


Figure 19: Sampling 5 potential pirates from two different pirate colleges. Pirate Training Unlimited (PTU) consistently produces average pirates (with scores between 40 and 60), while Unlimited Pirate Training (UPT), produces a wide range of pirates from 0 to 100.

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 19 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.

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 20 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] 1.6184811 0.6021107 0.7290129 1.6754286 0.6640214
```

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]  9.109339 10.427758  9.553366  6.413901 10.697239
```

```
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]
       )
```

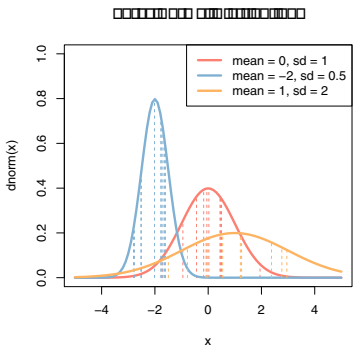


Figure 20: Three different normal distributions with different means and standard deviations.

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

```
## [1]  8.024336 14.333532  4.704237  4.201031 13.658394
```

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 20 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")
```

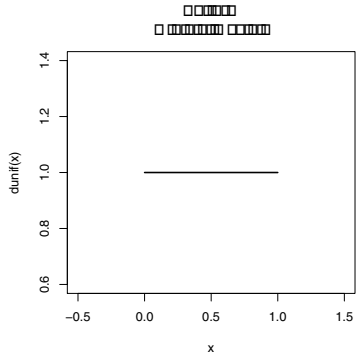


Figure 21: The Uniform distribution - known colloquially as the Anthony Davis distribution.

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] 0.64137201 0.57755064 0.96007265 0.05210931 0.08823298
```

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

```
## [1]  98.68427  26.83085 -24.17919  85.17760 -64.09378
```

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:

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

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.

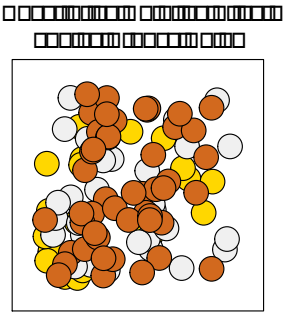
Drawing coins from a treasure chest

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`.

```
# 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" "gold"   "bronze" "bronze" "bronze" "gold"   "bronze"
##  [8] "bronze" "bronze" "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...

```
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))
       )
```



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

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

Figure 22: 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] 103.6144

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

## [1] 99.98045

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

```
## [1] 3.614376

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

## [1] -0.01954734
```

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 23: A gummy captain.

Additional Tips

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

- 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.

```
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.

9: Plotting: Part 1

Chapter Goals

1. High-level plotting commands: plot(), hist(), boxplot, barplot()
2. Main plotting parameters: main, xlab, ylab, xlim, ylim
3. Low-level plotting functions: abline(), points(), text(), legend()
4. Saving plots with pdf() and jpg()

Sammy Davis Jr. was one of the greatest 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.

But before we get to making some plots, let's go over some basics on specifying colors in R:

Figure 24: 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.

Color basics

There are many ways to specify colors in R – in the next chapter, I'll show you how to get really fancy with colors. But for now, let's cover two easy 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. Of course, all the basic colors are there, but there are tons of quirky colors like `snow`, `papayawhip` and `lawngreen`. To see all color names in R, run the code

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

plot(x = 1:11,
     y = rep(.5, 11),
     xlim = c(1, 11),
     ylim = c(0, 1),
     pch = 21,
     bg = gray(seq(0, 1, .1)),
     bty = "n",
     cex = 1.6,
     ylab = "",
     xlab = "",
     xaxt = "n",
     yaxt = "n"
     )

text(x = 5.5,
     y = .8,
     labels = "gray(x)",
```

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

`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.)

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 `beanplot()` 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

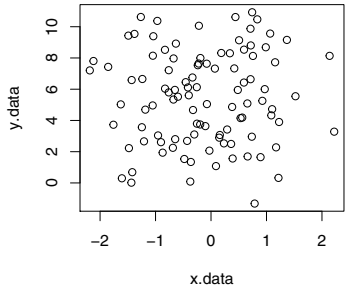
Most plotting functions have *tons* of optional arguments (also called parameters) that you can use to customize virtually everything in a plot. To see all of them, look at the help menu for `par` by executing `?par`. However, the good news is that you don't need to specify all possible parameters you 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.

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.

```
x.data <- rnorm(100, mean = 0, sd = 1)
y.data <- x + rnorm(100, mean = 0, sd = 1)

plot(x = x.data,
     y = y.data
     )
```

Scatterplot: `plot()`

The most common high-level plotting function is `plot(x, y)`. While its name sounds like it can make any kind of plot, the `plot()` function actually 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. Here are the main arguments to `plot()`:

If you only specify one vector of data (the x argument) to the `plot()` command, R will set all the y-values to 1.

plot()

`x, y`

Two vectors of data indicating the horizontal (x) values, and vertical (y) values of the points. If you don't specify the y argument, R will set the y value on all points to 1.

`type`

The type of plot. Use `type = "p"` for points (the default), `type = "l"` for lines, `type = "b"` for points and lines, and `type = "\n"` for no plotting. Using `type = "\n"` is helpful when you want to add plotting symbols later with low-level plotting functions.

`main, xlab, ylab`

String scalers indicating the text for the main plot title (`main`), the label for the horizontal x-axis (`xlab`), and the vertical y-axis (`ylab`)

`xlim, ylim`

A numeric vector of length two containing the minimum and maximum values of the x and y-axes. For example: `xlim = c(0, 100), ylim = c(50, 60)` will set the x limits to [0, 100] and the ylimits to [50, 60].

`col, bg`

The color (and possibly background) of the plotting symbols. For example `col = "red"` will create red symbols. If you use a symbol with a separate background (like `pch = 21`, `bg` controls the color of the background separately from the outline.

`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 filled circle, while `pch = "P"` will plot the character "P".

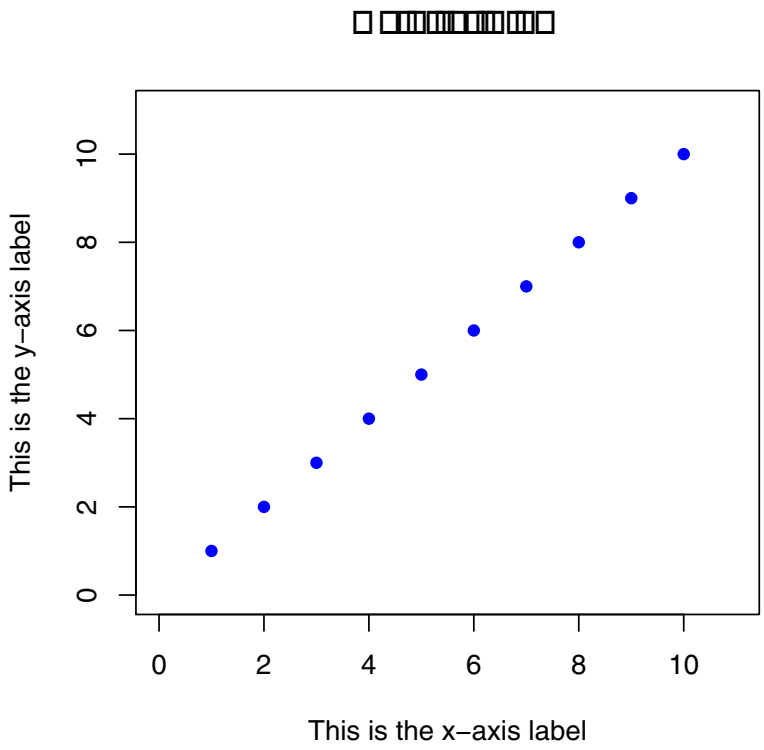
`cex`

A numeric vector specifying the size of the symbols (from 0 to Inf). The default size is 1. You can enter a vector of data here to change the size of the symbols based on some data (like a third variable).

As you can see, the `plot()` function, like many plotting functions,

has several optional arguments that allow you to change aspects of the plot. There are so many ways to customize the look of a plot that the number of optional arguments can be overwhelming at first. Let's start by looking at an example of a simple scatterplot showing ten data points: [1, 1], [2, 2] ... [10, 10].

```
plot(x = 1:10, # x-coordinates
     y = 1:10, # y-coordinates
     main = "My First Plot",
     xlab = "This is the x-axis label",
     ylab = "This is the y-axis label",
     xlim = c(0, 11), # Min and max values for x-axis
     ylim = c(0, 11), # Min and max values for y-axis
     col = "blue", # Color of the points
     pch = 16, # Type of symbol (16 means Filled circle)
     cex = 1 # Size of the symbols
     )
```



Aside from the x and y arguments, all of the arguments are optional. If you don't specify a specific argument, then R will use a default value, or try to come up with a value that makes sense. For example, if you don't specify the `xlim` and `ylim` arguments, R will set the limits so that all the points fit inside the plot.

```
     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
     )

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

Symbol types: pch

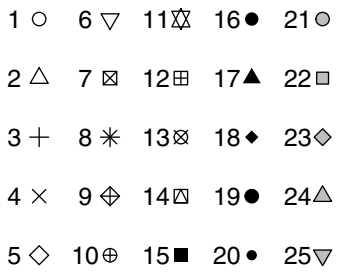


Figure 25: The symbol types associated with the pch plotting parameter.

When you create a plot with `plot(x, y)`, 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 25 in the margin.

Symbols differ in their border shape and how the filling is done. 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.

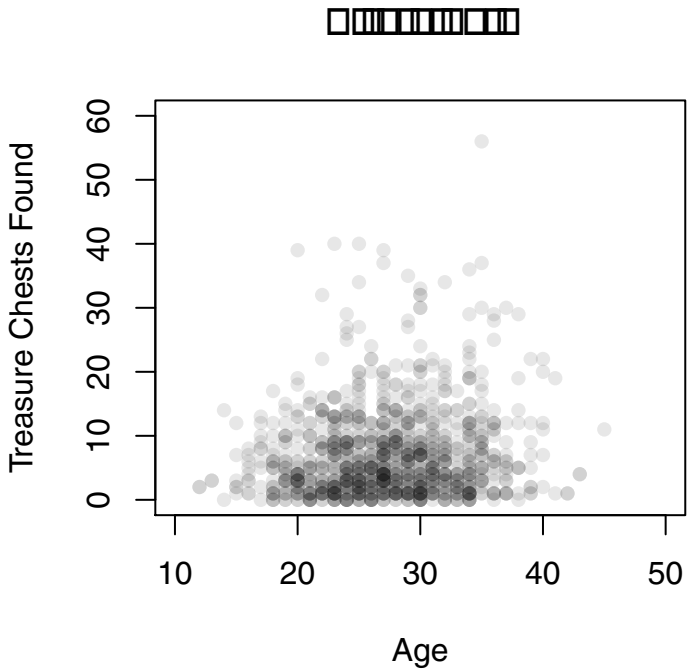
To specify the border color for all symbols, use the `col` argument. For symbols 21 through 25, you can additionally set the color of the fill using the `bg` ("background") argument.

Scatterplot of pirates data

In our first plot above, we just plotted the integers from 1 to 10. Of course, we can also plot vectors from columns in dataframes! Let's create a scatterplot from the pirates dataset. We'll put the pirate's age on the x-axis, and the number of treasure chests he/she has found on the y-axis. I'll add a few extra arguments like xlim and ylim to make the plot a bit wider than the default size. I'll also set the color of the points to be transparent gray (`col = gray(level = .1, alpha = .2)`) – this will show us where there are many points on top of each other.

As you can see in our plotting commands, we did not specify parameters like `xlim` and `ylim`. If you don't specify these arguments, R will try to come up with values that make sense given the data.

```
plot(x = pirates$age,
     y = pirates$tchests,
     xlab = "Age",
     ylab = "Treasure Chests Found",
     main = "Pirate data",
     pch = 16, # 24 are filled triangles,
     xlim = c(10, 50),
     ylim = c(0, 60),
     col = gray(level = .1, alpha = .1)
     )
```



It looks like most of the points are between the ages of 20 and 30 with between 0 and 10 treasure chests found!

Histogram: `hist()`

Histograms are the most common way to plot unidimensional numeric data. To create a histogram we'll use the `hist()` function:

hist()

`x`

A vector of numeric data

`breaks`

One of several values that defines how bins are created. The most common argument is a single number giving the number of bins you want in the histogram. See `?hist` for additional ways to specify this.

`col, border`

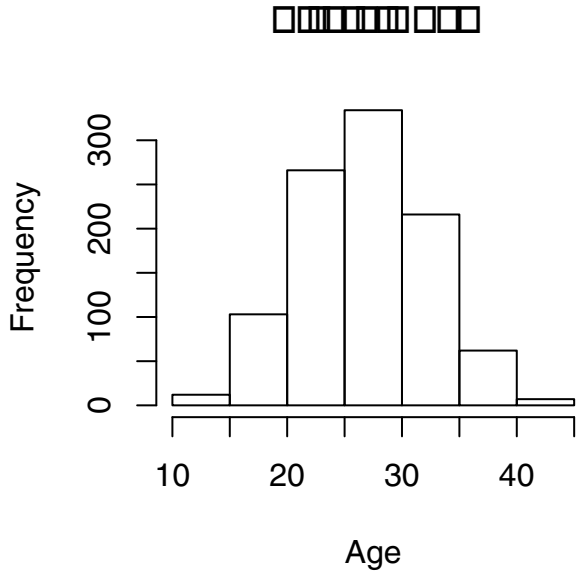
The color of the filling (col) and border (border) of the bars.

`main, xlab, ylab, xlim, ylim ...`

Other standard plotting arguments

Let's create a histogram of the ages of pirates in the `pirates` dataset:

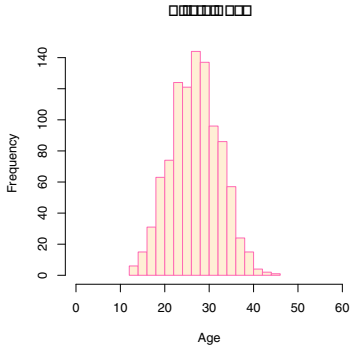
```
hist(x = pirates$age,
     main = "Pirate Ages",
     xlab = "Age",
     ylab = "Frequency"
     )
```




```
hist(x = pirates$age,
     main = "Pirate Ages",
     xlab = "Age",
     ylab = "Frequency",
     breaks = 20, # 20 Bins
     xlim = c(0, 60),
     col = "papayawhip", # Filling Color
     border = "hotpink" # Border Color
     )
```

We can get more fancy by adding additional arguments (see the margin figure)

You can see an example of a histogram in the margin Figure **??**.



The Pirate Plot: `pirateplot()`

The last high-level plotting function I want to show you is the `pirateplot()` function. The best part of a pirate plot is that it shows you three aspects of data: Raw data, Descriptive Statistics, and Inferential Statistics. Plus, as you can see in Figure **??**, they look much, much cooler than a boxplot or histogram. This means that you can quickly detect outliers, multiple modes, or missing data much better than you can with boxplots.

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

```
library(yarrr)
```

Here are some of the main arguments for `pirateplot()`. Check out the help menu (`?pirateplot`) to see several additional arguments

pirateplot()

`dv.name, iv.name`

Two string values indicating the names of the dependent variable and the independent variables in a dataframe.

`data`

A dataframe with columns corresponding to dv.name and iv.name. If you want to use a subset of data from a dataframe, just include the `subset()` function within this argument (e.g.; `data = subset(pirates, gender == "male")`)

`add.hdi`

A logical value indicating whether or not to add a 95% highest density interval (HDI) to the plot. This is a Bayesian interval which, assuming t-distributed data, gives you a 95% Bayesian posterior interval for the location of the population mean.

`labels`

An optional string vector specifying the names of groups (levels of the IV)

`my.palette`

A string (or vector of strings) indicating the colors of the beans. This can either be the name of a color palette in the piratepal function (run `piratepal(action = "p")` to see the names of all the palettes), or a vector of strings referring to colors. For example, `my.palette = "black"` will create a gray-scale plot.

`trans.vec`

A numeric vector of 5 values between 0 and 1 that indicate how transparent to make the colors in each distribution The five numbers correspond to the points, bean outlines, hdi band, the average line, and the white background respectively. For example, `trans.vec = c(1, 0, 0, 0, 0)` will only plot the raw data.

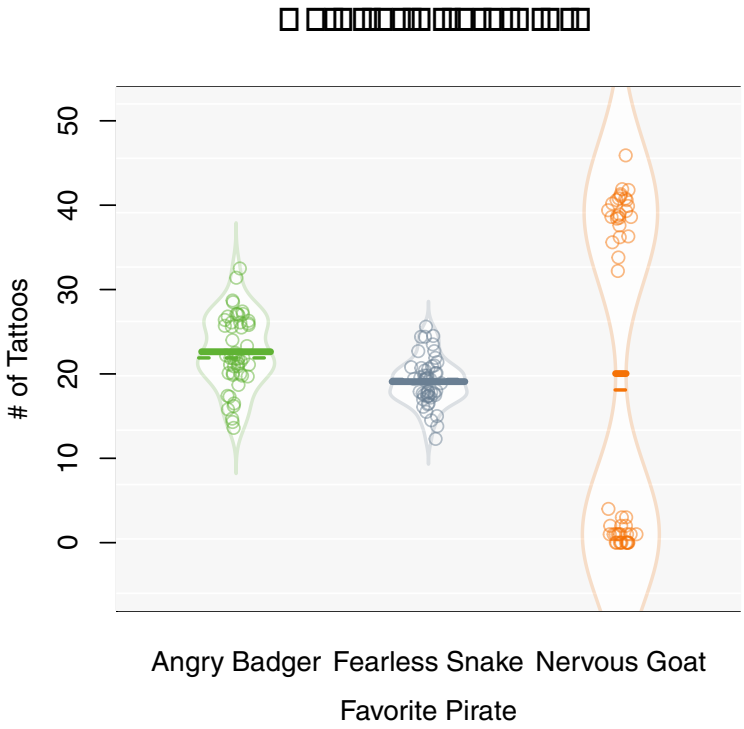
`...`

Other arguments passed on to the plot function (e.g.; main, xlab, ylab, ylim)

The code for creating pirateplots is a bit different from scatterplots with `plot()` and histograms with `hist()`. Let's start with an example:

I'll create a pirateplot using a new dataset called `BeardLengths`. This dataset shows the length of 100 pirates' beards on three different ships: the Angry Badger, the Fearless Snake, and the Nervous Goat:

```
library("yarrr")

pirateplot(dv.name = "Beard",
           iv.name = "Ship",
         data = BeardLengths,
         main = "My First Pirate Plot!",
         xlab = "Favorite Pirate",
         ylab = "# of Tattoos"
         )
```



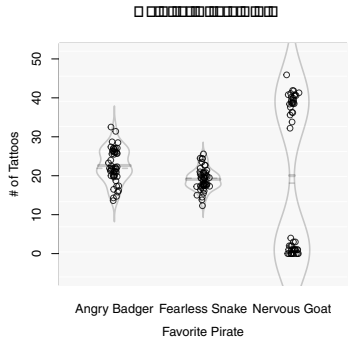
Let's walk through the main arguments. First, in the `dv.name` and `iv.name` arguments, we specify the names of the dependent and independent variables in the dataset that we want to plot as strings. Next, in the `data` argument we specify the dataframe object that contains the data. Finally, we can specify some plotting labels like `main`, `xlab` and `ylab` just like in `plot()`.

Other plotting types

Compared to pirate plots, these next two plotting types, barplots and boxplots, are pretty boring. But if you want to make them, here's

You can can easily customize the look of your pirate plot with additional arguments. For example, to make a black and white plot with a stronger focus on the raw data, set `my.palette = "black"` and `trans.vec = c(.2, .8, .8, .8, .8, .8)`:

```
library("yarrr")

pirateplot(dv.name = "Beard",
           iv.name = "Ship",
         data = BeardLengths,
         my.palette = "black",
         trans.vec = c(.2, .8, .8, .8, .8, .8),
         main = "My First Pirate Plot!",
         xlab = "Favorite Pirate",
         ylab = "# of Tattoos"
         )
```



how:

Barplot: barplot()

A barplot shows a single bar representing a single value for every level of some independent variable. Here are the main arguments to `barplot()`

As always, check the help menu for barplots with `?barplot` for additional arguments and examples

`barplot()`

`height`

A numeric vector indicating the heights of the bars

`names.arg`

An optional string vector indicating the names under the bars

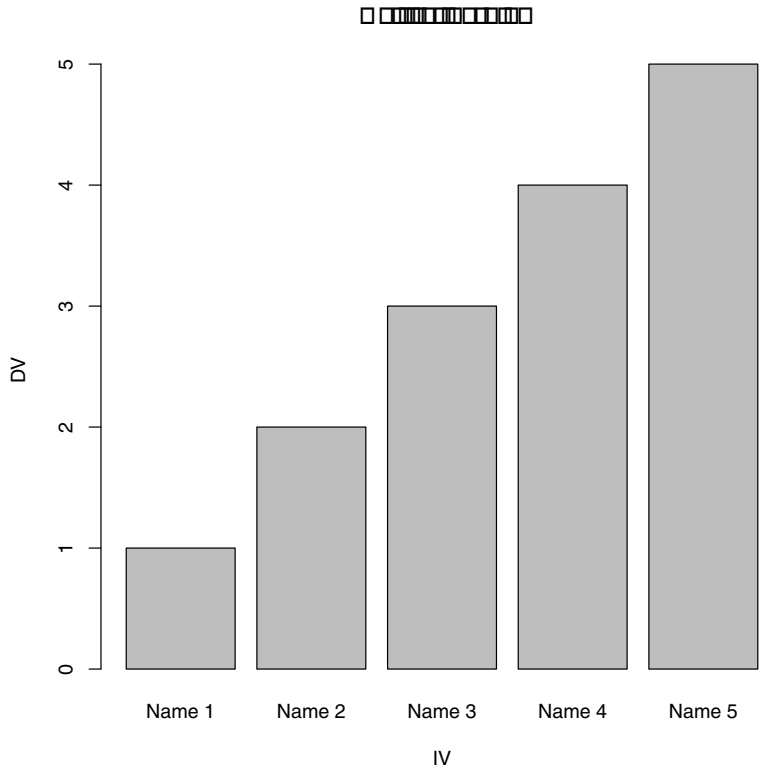
`border, col`

The color of the borders (`border`) and filling (`col`) of the bars.

`main, xlab, ylab`

Optional string values indicating the main and sub-titles, and axis labels of the plot

```
barplot(height = 1:5,
        names.arg = c("Name 1", "Name 2", "Name 3", "Name 4", "Name 5"),
        main = "My first barplot",
        xlab = "IV",
        ylab = "DV"
        )
```



Boxplot: boxplot()

Boxplots aren't used so often anymore, but if you want to be old-school, they're a good place to start. To create a boxplot, use the `boxplot()` function:

boxplot()

`formula, data`

A formula in the form `formula = dv ~ iv` indicating the dependent variable and independent variable, and a dataframe containing the variables in the formula. For example `formula = height ~ sex`

`subset`

An optional logical vector indicating a subset of the data to plot. For example, the command `subset = gender == "male" & weight < 120`, will only plot data for males with weight less than 120.

`border, col`

The color of the borders (`border`) and filling (`col`) of the boxes.

`names`

A string vector indicating the names of the boxes. E.g.; `names = c("males", "females")`

When you use `boxplot()`, you can either specify a single vector of data to plot, or you can use a `formula` to indicate a dependent and independent variable. If you do this, R will add separate boxes for all values of the independent variable.

Let's go through two examples of boxplots in Figure **??**. In the first plot, I just entered a single vector of data: `pirates$age` representing all weight data in the dataframe. In the second plot, I plotted separate boxes for the different levels of `college` using the formula notation `age ~ college`. If you're wondering how R knows that I'm referring to the `pirates` dataframe when using the formula notation, the answer is that I had to specify the name of the dataframe `pirates` as an additional `data` argument. This argument tells R that the objects in the formula are names in the `pirates` dataframe.

```
boxplot(x = pirates$age,
        names = "All Data",
        ylab = "Age",
        main = "Plot 1: All Ages")
```

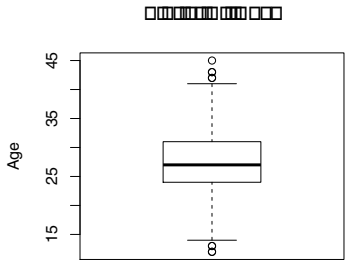


Figure 26: Plotting data from a single vector

```
boxplot(age ~ college, # Formula: DV ~ IV is Diet
        data = pirates,
        xlab = "college",
        ylab = "Weight",
        main = "Plot 2: Age separated by college")
```

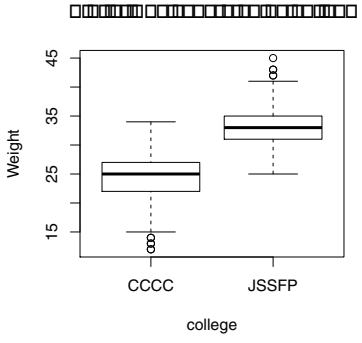


Figure 27: Plotting data as a function of levels of an independent variable using the y ~ x formula notation.

Low-level plotting functions

Once you've created a plot with a high-level plotting function, you can add additional elements, like additional data points, reference lines, text, and legends using low-level plotting functions. There are many low-level plotting functions, I will focus on those that I frequently use.

Starting with a blank plot

I like using low-level plotting functions so much that I frequently like to start with a (mostly) blank plotting space, and then add the main plot elements using low-level plotting functions. To start with a blank plot, use the `plot()` function combined with the arguments `type = "n", xaxt = "n", yaxt = "n"` and all labels set to `""`. See margin Figure 28 for an example

Once you've created a blank plot, you can proceed to add all the elements you'd like with low-level plotting commands. Let's start with `points()`, which adds points to an existing plot

```
# Create a blank plot
plot(x = 1,
     xlab = "",
     ylab = "",
     xaxt = "n", yaxt = "n",
     type = "n",
     xlim = c(0, 100), ylim = c(0, 100),
     main = "Blank Plot")
```

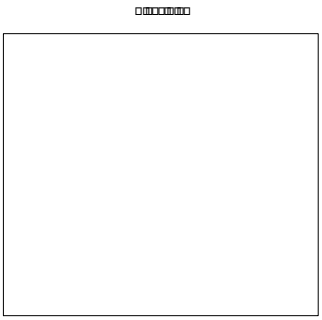


Figure 28: A blank plot. Useful to start with before adding elements with low-level plotting commands. Just make sure to set the axis limits to values that make sense for your future data.

points()

points()

`x, y`

Two vectors corresponding to the x and y values of the points

`pch, col, bg`

Type of plotting symbols (`pch`), color of the plotting symbols (`col`), and the color of the filling of the plotting symbols ((`bg`)) for plotting symbols 21 through 25

For example, to add red circle points to a plot where `x.vals` are the x-values and `y.vals` are the y.values, you can run the code:

```
points(x = x.vals, # x-values
       y = y.vals, # y-values
       col = "red", # Symbol color
       pch = 16 # Symbol type (circles)
       )
```

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!

In margin Figure 29, I use the points function to plot data from pirates, where pirates from Captain Chunk's Canon Crew are plotted

```
# Get subsets of data
college.1 <- subset(pirates, college == "CCCC")
college.2 <- subset(pirates, college == "JSSFP")

# Create a blank plot
plot(x = 1,
     xlab = "Age",
     ylab = "Number of Tattoos",
     type = "n",
     main = "Pirate Tattoos by college",
     xlim = c(1, 50),
     ylim = c(0, 20))

# Add red points for college 1
points(x = college.1$age,
       y = college.1$tattoos,
       pch = 16,
       col = "red")

# Add skyblue points for diet 2
points(x = college.2$age,
       y = college.2$tattoos,
       pch = 16,
       col = "skyblue")
```

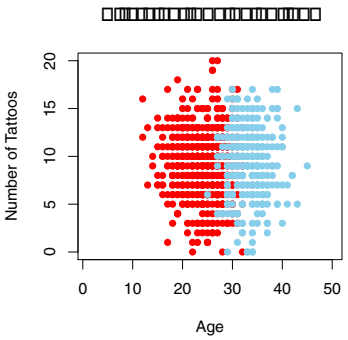


Figure 29: Adding additional points to an existing plot with points()

in red, and pirates from Jack Sparrow's School of Fashion and Piratry are plotted in skyblue.

Next, we'll look at `abline()` which adds straight lines to a plot:

abline()

abline()

`a, b`

Numeric scalars or vectors indicating the slope (`a`) and intercept `b` of the line(s)

`h, v`

Numeric scalars or vectors indicating the y-value of horizontal lines (`h`) or x-values of vertical lines `v`. For example, `abline(h = 1)` will add a horizontal line at y = 1, while `abline(v = 10)` will add a vertical line at x = 1

`lty, lwd`

Type (`lty`) and width (`lwd`) of line. See margin Figure to see line types.

For example, to add a vertical line at an x-value of 0 or a horizontal line at a y-vale at 100 you'd enter

```
abline(v = 0) # Add a vertical line at x = 0
abline(h = 100) # Add a horizontal line at y = 100
```

You can easily use abline() to add gridlines to plots by entering vectors in the `h` and `v` arguments. For example, to add gridlines to a plot at x-values and y-values from 0 to 10 in steps of 1, you'd enter

```
abline(v = 1:10) # Add vertical lines from 1 to 10
abline(h = 1:10) # Add horizontal lines from 1 to 10
```

In margin Figure 31 I add gridlines and a diagonal reference line to a plot before adding points.

Next, we'll move on to `text`, which adds text to a plot

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

```
par(mar = c(3, 0, 6, 0))
plot(1,
     xlim = c(0, 7),
     ylim = c(0, 1),
     type = "n",
     xlab = "lty values",
     ylab = "",
     xaxt = "n",
     yaxt = "n",
     bty = "n",
     main = "")

abline(v = 1:6,
       lty = 1:6,
       lwd = 2)

mtext(1:6,
      side = 3,
      at = 1:6,
      cex = 1.5,
      line = 1)

mtext("lty = ...",
      side = 3,
      at = 3.5,
      line = 4,
      cex = 2)
```

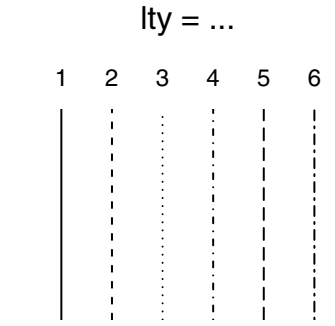


Figure 30: Line types generated from arguments to lty.

```
# Create a blank plot
plot(x = 1,
     xlab = "Group",
     ylab = "Length",
     type = "n",
     main = "Gridlines with abline()",
     xlim = c(0, 10), ylim = c(0, 10))

# Add horizontal gridlines
abline(h = 1:10,
       lwd = 1,
       col = gray(.8))

# Add vertical gridlines
abline(v = 1:10,
       col = gray(.8))

# Add main diagonal reference line
abline(a = 0,
       b = 1,
       lwd = 2,
       lty = 2)

# Create data
x.data <- rnorm(100, mean = 5, sd = 2)
y.data <- x.data + rnorm(100, mean = 0, sd = 3)

# Add points
points(x = x.data,
       y = y.data, pch = 16,
       col = gray(.4, alpha = .5),
       cex = c(runif(90, 0, 2), runif(10, 3, 4))
       )
```

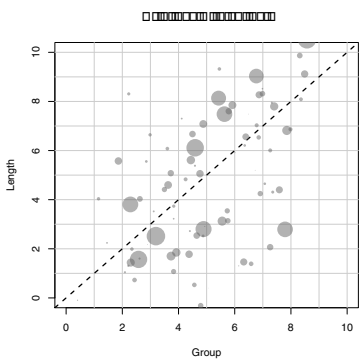


Figure 31: Adding gridlines to a plot

(like a third variable) for every point in a plot. Here are the main arguments to `text()`

text()

`x, y`

Numeric scalars or vectors specifying the coordinates of the labels

`labels`

String vector of the text you're plotting. Use the paste() function to create multiple strings or combine strings with numeric objects.

`cex`

Numeric scalar or vector specifying the size of the `labels`

`adj`

A numerical value between 0 and 1 specifying the horizontal and/or vertical justification of text. Use 0 for left justification, .5 for centering, and 1 for right justification.

`pos`

Specifies the position of the text relative to the x-y coordinates. Values of 1, 2, 3 and 4 respectively indicate below, to the left, above, and to the right of the x-y coordinates.

`font`

The font face. 1 = plain, 2 = bold, 3 = italic, 4 = bold-italic.

For example, if you want to add the text "This is the center of the plot" to a plot at the coordinates (0, 0), you'd enter

```
text(x = 0,
     y = 0,
     labels = "This is the center of the plot")
```

Alternatively, let's say you have a scatterplot and wanted to add the x-values in text right above (pos = 3) each point, you could do this by using the code:

```
text(x = x.data, # X-values of data
     y = y,data, # X-values of data
     labels = x.data, # Add text of the x-values
     pos = 3 # Put the text right above the points
```

```
    )
```

To see `text()` in action, look at margin Figure 32 where I put the x-values of some random data right above their points:

When entering text in the `labels` argument, keep in mind that R will, by default, plot the entire text in one line. However, if you are adding a long text string (like a sentence), you may want to separate the text into separate lines. To do this, add the text `"\n"` where you want new lines to start. Look at Figure for an example.

Formatting text for plotting

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:

paste()

`...`

One or more scalars or vectors (numeric or string) to be combined. For example `paste("The mean of x is ", mean(x), sep = "")` will create a string combining text and a statistic calculated from data.

`sep`

A character string that separates the arguments. Set to `""` for no separation

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 <- rnorm(200, mean = 20, sd = 10)
mean(data)

## [1] 21.21807

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

```
# Step 1: Generate Data

x.data <- rnorm(20, mean = 0, sd = 20)
y.data <- x.data + rnorm(20, mean = 0, sd = 20)

# Step 2:  Create a blank plot

plot(x = 1, xlab = "", ylab = "",
     type = "n", main = "Adding text with text()",
     xlim = c(-50, 50), ylim = c(-50, 50))

# Step 3: Add points

points(x = x.data, y = y.data,
       pch = 16, col = gray(.5, alpha = .2))

# Step #4: Add x-coordinates in text above points
text(x = x.data,
     y = y.data,
     labels = round(x.data, 0),
     pos = 3, # put coordinates below the points
     cex = .7
     )
```

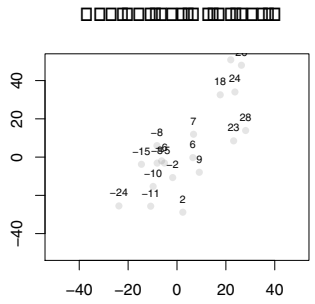


Figure 32: Adding text to a plot with text().

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"
     )
```

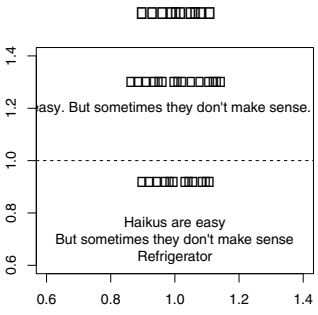


Figure 33: Using the `"\n"` tag to plot text on separate lines.

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.

```
## [1] "The mean of the group is 21.218069485714"

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

## [1] "The mean of the group is 21.22"
```

You can also use vectors as arguments to the `paste()` function. For example, let's say that you want to create a vector of labels for 5 groups, and you want each group to be labelled `"Group X"`. We can easily do this with `paste()`

```
paste("Group", 1:5, sep = " ")

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

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 34, 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"
       )
```

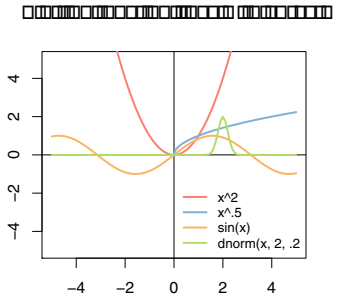


Figure 34: Using `curve()` to easily create lines of functions

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"
       )
```

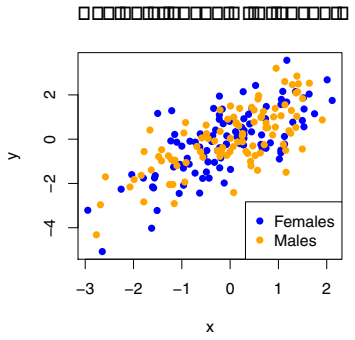


Figure 35: Creating a legend labeling the symbol types from different groups

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.

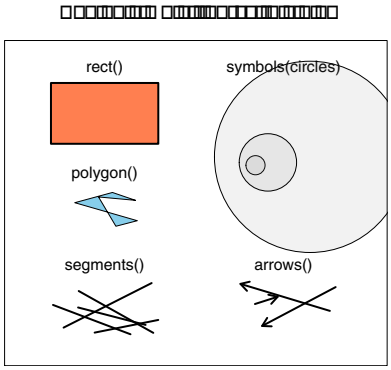
`axis()`

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.

```r
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/Nathaniel/Desktop/plot.pdf"` when creating a pdf, and `file = "/Users/Nathaniel/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 = "/Users/Nathaniel/Desktop/My Plot.pdf",   # The directory you want to save the file in
    width = 4, # The width of the plot in inches
    height = 4 # The height of the plot in inches
    )

# Step 2: Create the plot
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.

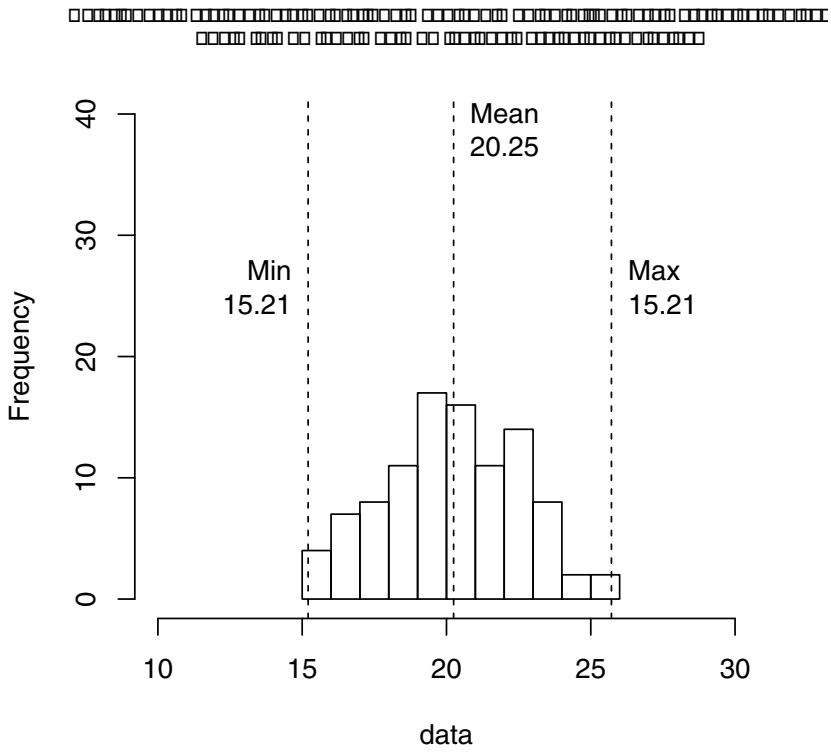
A worked example: Creating a plot with automated numeric labels

Let's use the `paste()` command to create a histogram with labels indicating the mean, median, min, and mean of the dataset. We'll do this in 5 steps

1. Generate the data and the histogram
2. Add text and reference line for the mean
3. Add text and reference line for the minimum
4. Add text and reference line for the maximum
5. Add a subtitle in full sentences with each summary statistic.

```
# Step 1: Generate data and main histogram
data <- rnorm(100, mean = 20, sd = 2)

title.text <- paste(
  "Note: There were ", length(data), " data points. The mean and median of the data were ",
  round(mean(data), 2), " and ", round(median(data, 2)), ".\nThe minimum and maximum values were ",
  round(min(data), 2), " and ", round(max(data), 2), ".", sep = "")


hist(data,
     xlim = c(10, 30),
     ylim = c(0, 40),
     main = title.text,
     cex.main = .7
     )

# Step 2: Add mean text and line
text(mean(data), 38,
     labels = paste("Mean\n", round(mean(data), 2), sep = ""),
     adj = 0,
```

```
     pos = 4
)
abline(v = mean(data), lty = 2)

# Step 3: Add minimum text and line
text(min(data), 25,
     labels = paste("Min\n", round(min(data), 2), sep = ""),
     adj = 0,
     pos = 2
)

abline(v = min(data), lty = 2)

# Step 4: Add maximum text and line
text(max(data), 25,
     labels = paste("Max\n", round(min(data), 2), sep = ""),
     adj = 0,
     pos = 4
)

abline(v = max(data), lty = 2)
```



The benefit of using `paste()` over hard-coding the text (for example by typing `labels = "Mean = 20"`) is that the code will automatically change the value of the mean when the data changes. To see this in action, run the code above several times and see how the numbers automatically update. In later chapters, when use loops to

create multiple graphs over different sets of data, this will become extremely helpful!

Additional Tips

- Many high-level plotting functions can be used like low-level plotting functions if you add an additional argument like `new = F` or `add = T`. Look at the help menu for specific high-level functions to see which arguments allow you to add a high-level plot to an existing plot.

10: Plotting: Part Deux

1. Specifying and creating colors
2. Specifying plot margins with `par(mar)`
3. Putting several plots together with `par(mfrow)` and `layout`
4. Using different fonts in plots

Colors in R

There are many ways to specify colors in R. If you want to specify a color directly, you can do that in one of the following ways:

Specifying colors as a string

The easiest way to specify a color is to just write its name as a string. For example, you can write `"blue"`, `"lightgreen"`, `"red"`, among many other colors. To see a list of all the named colors, look at the vector `colors()` which contains all 657 named colors in R. Here is a random sample of 10 of them (to see all the colors, look at the color graph in the Appendix)

```
colors()[sample(1:length(colors()), 10)]
```

```
##  [1] "gray88"      "grey16"      "deeppink4"   "purple4"     "darkorchid1"
##  [6] "mistyrose4"  "brown4"      "honeydew3"   "khaki1"      "darkred"
```

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 36.

```
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"
     )
```

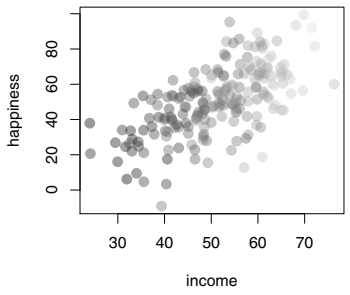


Figure 36: Using the `gray()` function to easily create shades of gray in plotting symbols based on numerical data.

RGB values: `rgb()`

Every color can be defined by its RGB ("Red", "Green", "Blue") value. This value specifies the combination of shades of Red, Green and Blue that create that color. Traditionally, each color shade is defined on a scale from 0 to 255. For example, the RGB value [255, 0, 0] is pure Red, while [0, 255, 0] is pure Green.

To create a color from RGB values, use the function `rgb()`

`rgb()`

`red, green blue`

Numeric arguments indicating the strength of red, green, and blue hues

`maxColorValue`

A number indicating the maximum possible hue value. The default is 1 - however, most people use `maxColorValue = 255`

`alpha`

The opacity of the color(s) inputted as a number between 0 and `maxColorValue`

When you use the function `rgb()`, the function will return a string as output. The string will look like nonsense to you, but that's just how R names colors:

```
rgb(red = 0, green = 255, blue = 0, maxColorValue = 255) # pure red

## [1] "#00FF00"

rgb(red = 0, green = 255, blue = 0, alpha = 100, maxColorValue = 255) # transparent red

## [1] "#00FF0064"

rgb(red = 100, green = 100, blue = 100, maxColorValue = 255) # even mixture

## [1] "#646464"
```

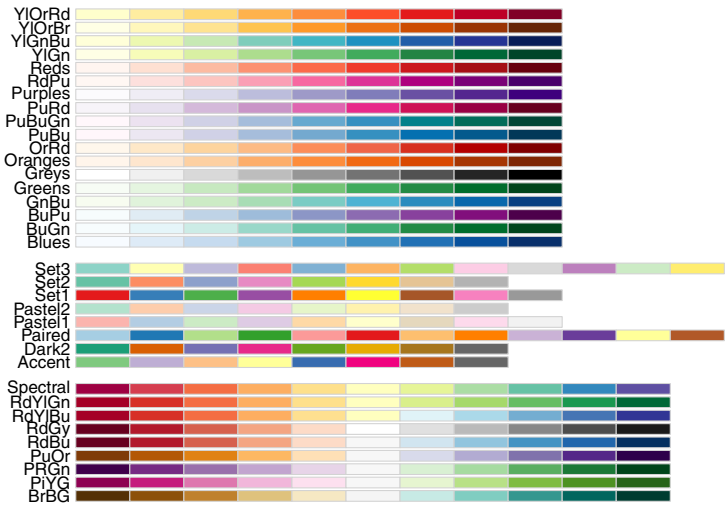
Now, if you're not used to interpreting colors as RGB values, you may gloss over this section and think you'll never use it. However, look at this chapter's *Additional Tips* for a really cool method of 'stealing' the exact color from *anything* on your computer screen and using `rgb()` to then use that color in your R plots!

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.

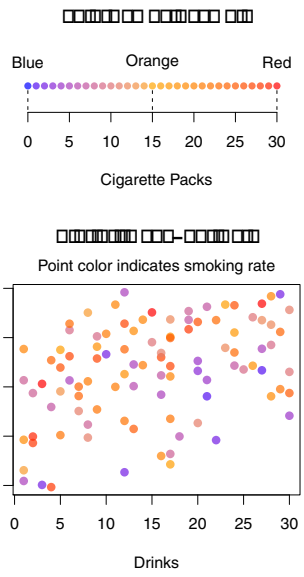
`colorRamp2` allows you to do exactly this. The function has three arguments:

- `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

```
require("RColorBrewer")
require("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` which takes a number as an argument, and returns the corresponding color:

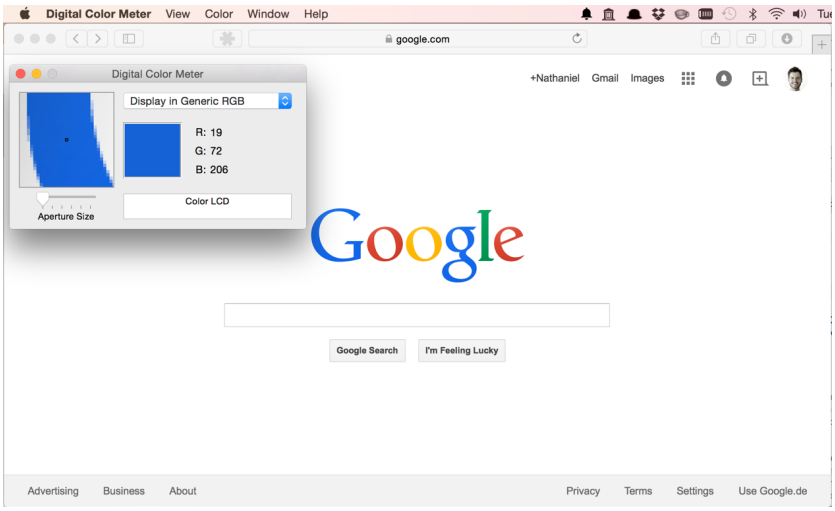
```
smoking.colors <- colorRamp2(breaks = c(0, 15, 30),
                  colors = c("blue", "orange", "red"),
                  transparency = .3
                  )

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)
```



Plot margins

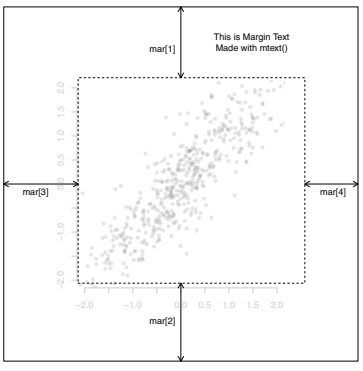
All plots in R have margins surrounding them that separate the main plotting space from the area where the axes, labels and additional text lie.. To visualize how R creates plot margins, look at margin Figure .

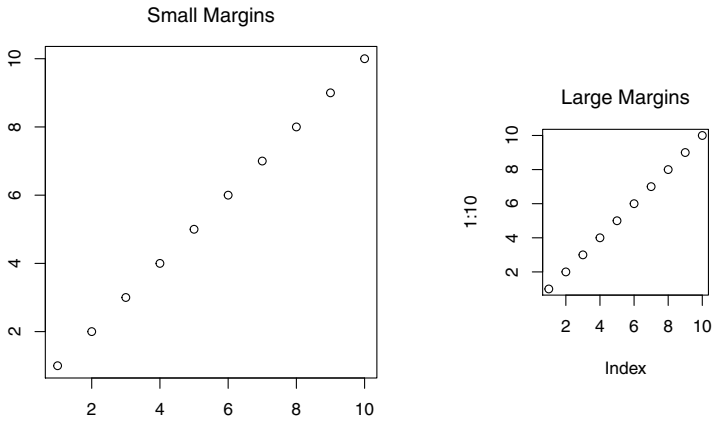
You can adjust the size of the margins by specifying a margin parameter using the syntax `par(mar = c(a, b, c, d))` before you execute your first high-level plotting function, where a, b, c and d are the size of the margins on the bottom, left, top, and right of the plot. Let's see how this works by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

```
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)`

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.

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)

}
```

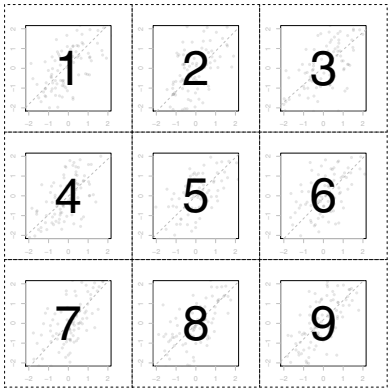


Figure 38: A matrix of plotting regions created by `par(mfrow = c(3, 3))`

- `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 39)

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)
```

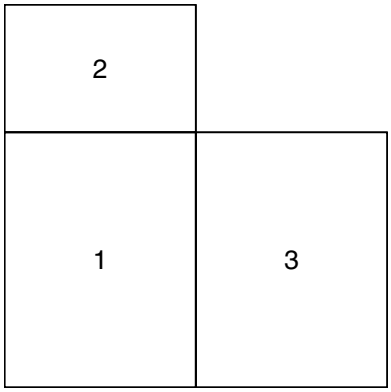
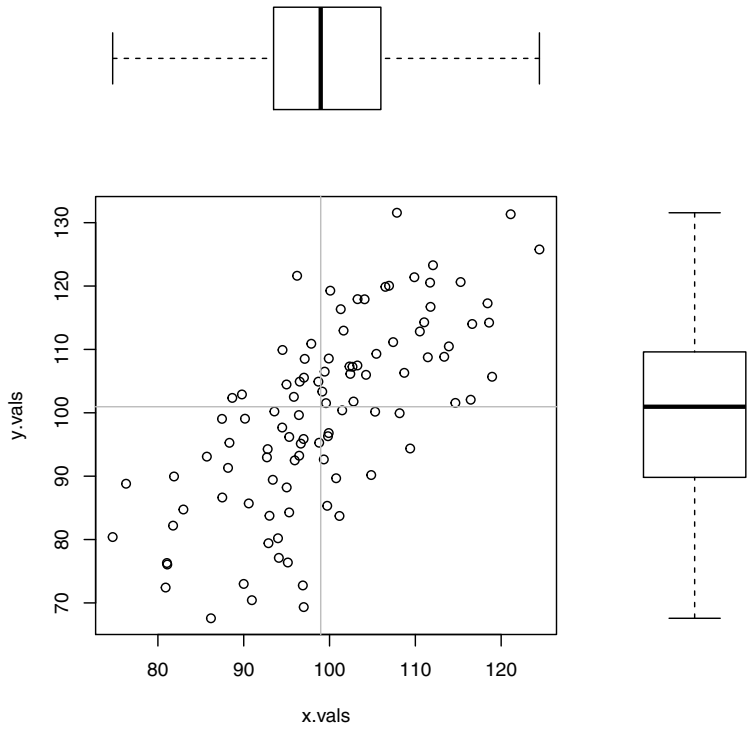


Figure 39: A plotting layout created by setting a layout matrix and specific heights and widths.

```
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)
```



Using alternative fonts in pdfs with the extrafont package

If you don't like the default font that R uses in creating plots, you can use the `extrafont` package to use additional fonts in plots saved as .pdf files. However, let me warn you that it's not as easy as just selecting a font from a drop-down menu (like in Word). To use other fonts, follow these four steps:

First, install and load the `extrafont` package

```
install.packages("extrafont")
library("extrafont")
```

```
## Registering fonts with R
```

Second, import the fonts on your computer into R by running the `font_import()` function. When you execute this, you'll receive a warning telling you that importing the fonts may take a few minutes. Type "y" and watch R do it's magic. Don't worry if you see some warnings or if it takes a few minutes,once you've run `font_import()` once you *won't* need to run it again on your machine.

```
font_import()
```

Third,load your fonts into your current R session using the function `loadfonts()`. Unfortunately, you DO need to run this in each R session.

```
loadfonts()
```

Now you're ready to go. To see which fonts are available to use in your plots, use the `fonts()` function. When you execute this function, you'll see a table with all the fonts you can use. Let's do this on my system (I'll just print the first 50 values here)

```
fonts()[1:50] # Show me the first 50 fonts on my system
```

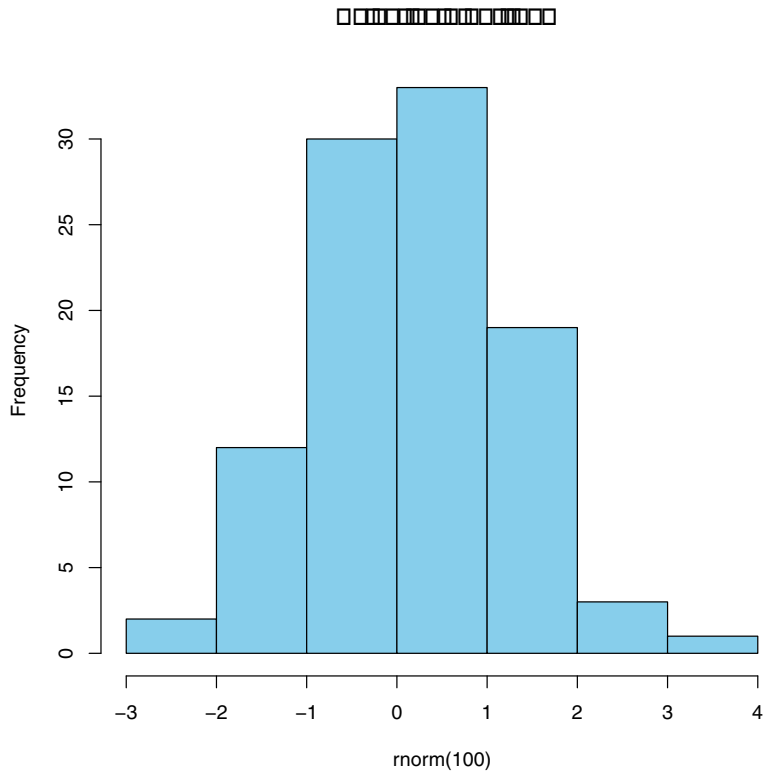
```
## NULL
```

You will likely have more or less fonts than I have on my system - if you want more fonts, you'll need to download them. Now that we have a list of fonts we can use, we can finally create a plot using the new font. To do this, we need to add two special arguments when creating our plot:

1. In the `pdf()` function, add the argument `family = "fontname"`, where `fontname` is the name of the font you want to use.
2. After executing `dev.off()` to finish the plot, execute the command `embed_fonts("filename.pdf")`. This command will embed the font in the pdf file.

Let's follow these steps to create a plot using the Helvetica Light font. Again, I found this font on my computer by running `fonts()`. If you don't have this font on your computer, then it won't work for you. Instead, replace the argument `"HelvLight"` with a different font on your system:

```
pdf("/Users/Nathaniel/Dropbox/Git/YaRrr_Book/media/helveticalight.pdf",
    width = 4, height = 4,
    family = "HelvLight" # Specify the font in the plot
    )

## Error in pdf("/Users/Nathaniel/Dropbox/Git/YaRrr_Book/media/helveticalight.pdf",
: unknown family 'HelvLight'

hist(x = rnorm(100), # some random data
     col = "skyblue",
     main = "Helvetica Light font")
```




```
dev.off()

## null device
##           1

embed_fonts("/Users/Nathaniel/Dropbox/Git/YaRrr_Book/media/helveticalight.pdf" # the fonts in the pdf
```

Look at Figure X to see the plot that this code created

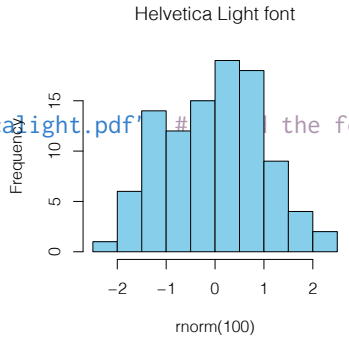
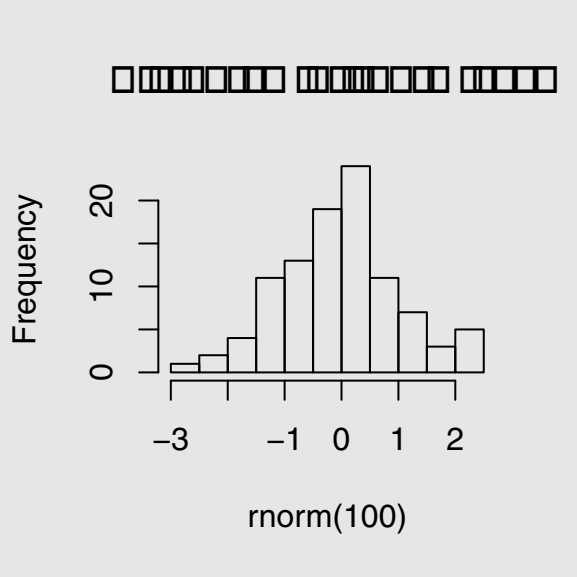


Figure 40: Plot created with Helvetica Light font (see the main text for plotting code).

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:

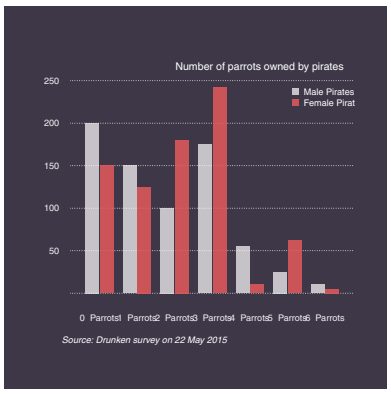
```
par(bg = gray(.9))
hist(x = rnorm(100))
```



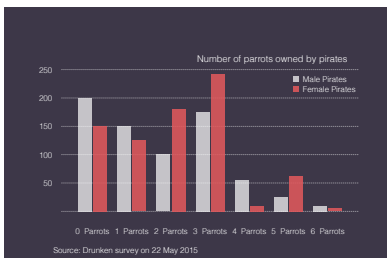
See Figure 41 for a nicer example.

- 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/Nathaniel/Dropbox/Git/YaRrr_Book/media/parrothelvetica.pdf",
    width = 8, height = 6, family = "HelvLight")


## Error in
pdf("/Users/Nathaniel/Dropbox/Git/YaRrr_Book/media/parrothelvetica.pdf",
: unknown family 'HelvLight'


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()


## null device
##           1


embed_fonts("/Users/Nathaniel/Dropbox/Git/YaRrr_Book/media/parrothelvetica.pdf") #
```



11: 1 and 2-sample Null-Hypothesis tests

Chapter Goals

1. Learn about hypothesis test objects in R
2. One and two sample tests: Correlations, t-tests and chi-square
3. Use the apa function to easily get APA-style conclusions

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 nipple rings more likely to wear bandannas than those without nipple rings? Glad you asked, let's see how we can answer these questions some hypothesis tests.

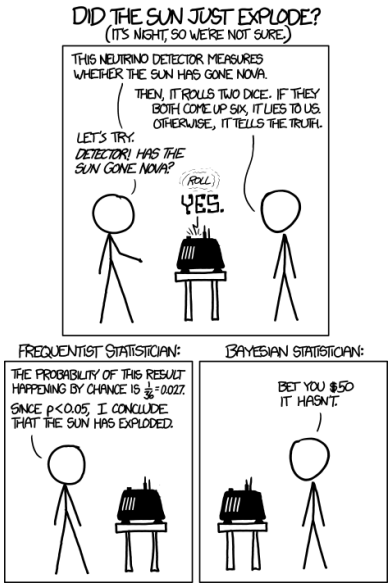


Figure 42: xkcd comic. Currently used without any permission.

Warning about null-hypothesis tests with "frequentist" statistics

Until recently, null-hypothesis testing using frequentist statistics has been the most popular method of conducting inferential statistics. However, it has serious flaws. While I can't go into the details here, I can point out that the main flaw is that frequentist statistics don't give you the information you really want to know. For example, imagine that you are comparing the effectiveness of a cancer drug to a placebo. After conducting a double-blind study, where you give some patients the placebo and some patients the drug, you want to know the probability that that the drug is better than a placebo. Unfortunately frequentist statistics cannot give you this information. They can only tell you the probability of getting a specific result *given* that the null hypothesis (in this case, that the drug is equally as effective as the placebo) is true. If that sounds confusing, it's because it is. A better alternative is Bayesian statistics which *can* give you posterior probability information. Unfortunately, Bayesian statistics can be computationally demanding, so in the past we've lived with frequentist statistics and tried to ignore its fundamental flaws. However, given improvements in processing speed, we can

now easily conduct Bayesian alternatives to frequentist tests on modern computers.

We will cover Bayesian statistics in Chapter X and I strongly encourage you to adopt them in your own analyses. However, for the purposes of completeness, I'll show you how to conduct most of the standard frequentist tests here.

T-test

We use t-tests to compare the sample mean of data to some hypothesized mean. In a one sample test, we use one set of observations to test whether or not the population mean is different from a hypothesized value. In a two-sample test, we use two sets of observations to test whether or not the two populations have different means.

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`).

`t.test(x, y)`: Conduct either a one sample t-test on a vector x, or a two-sample t-test on two vectors x and y.

There are two ways to use the `t.test()` function. The first way is to enter one (or more) vectors as arguments to the function as follows:

t.test(x, y)

`x, y`

Either one vector of data (x) for a one-sample t-test, or two vectors (x, y) for a two-sample test.

`alternative`

A character string indicating whether the test is two-tailed or one-tailed (including the direction). Type "t" for two-tailed, "g" for a 'greater than' one-tailed test, or "l" for a "less than" one-tailed test.

`mu`

he population mean under the null hypothesis.

`paired, var.equal`

`paired`: A logical value (either T or F) indicating whether the test is paired (T) or unpaired (F). Only use this for two-sample tests.

`var.equal`: A logical value indicating whether or not you treat the two variances as equal.

Let's do an example using the pirate survey dataset. If you haven't

downloaded the pirate survey dataset, check Chapter 1 for instructions.

One-Sample t-test

The format for a one-sample t-test is as follows:

```
t.test(x = data,  # A vector of data
       mu = 0, # The null hypothesis
       alternative = "t" # Two tailed test (use "l" or "g" for one-
       )
```

where x is a vector of data, mu is the population mean under the null hypothesis, and alternative is ‘‘t" for a two-tailed test, or “l" or ‘‘g" for a one-tailed test.

Let’s do a one-sample t-test on the age of pirates in our survey. Specifically, let’s see if the average age of the pirates is significantly different from 20. In this case, our vector x is pirates$age:

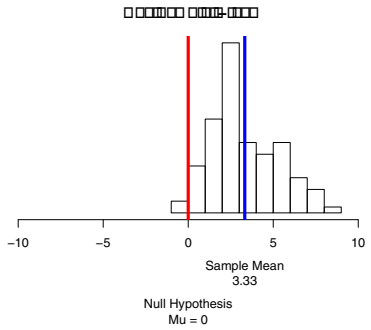
```
test.result <- t.test(x = pirates$age, # Vector of data to test
                      mu = 20,   # Null hypothesis is mean = 30
                      alternative = "t" # Two-tailed test
                      )
```

You’ll notice that when you assign the t.test to an object (in this case we called it test.result), you do not see any output. To see the output of the test, you need to tell R to print the object by executing the name of the test object:

```
test.result  # Print the results of the t.test

##
##  One Sample t-test
##
## data:  pirates$age
## t = 42.243, df = 999, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 20
## 95 percent confidence interval:
##  27.05147 27.73853
## sample estimates:
## mean of x
##    27.395
```

Now, you can see the main output of the test. Looks like we got a test statistic of 42.24 and a resulting p-value that’s pretty darn small. But what if you want to access specific values like the test statistic or the p-value? Thankfully, this is easy in R. To see which information you can extract from the t-test object, apply the names() function to the test object:

```
par(mar = c(10, 0, 3, 1))
plot(1, xlim = c(-10, 10), ylim = c(0, 1),
     yaxt = "n", main = "One Sample t-test",
     type = "n", bty = "n", ylab = "", xlab = "")

samples <- rnorm(100, mean = 3, sd = 2)

par(new = T)
hist(samples, yaxt = "n", xaxt = "n", xlab = "",
     ylab = "", main = "", xlim = c(-10, 10))

mtext(paste("Sample Mean\n", round(mean(samples), 2), sep = ""),
      side = 1, line = 3.5, at = mean(samples))

abline(v = mean(samples), lty = 1, col = "blue", lwd = 4)

mtext("Null Hypothesis\nMu = 0", side = 1, line = 6, at =0)
abline(v = 0, lty = 1, col = "red", lwd = 4)
```



If you want to see what information is in a test object, just apply names() to the object. You can then extract specific information with $

```
names(test.result)

## [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
## [6] "null.value"  "alternative" "method"      "data.name"
```

From this vector of names, I see that I can extract the test statistic with the name `statistic` and the p-value with the name `p.value`. To get these from the test object, use the $ operator:

```
test.result$statistic  # Show me the test statistic

##        t
## 42.24292

test.result$p.value # Show me the p.value

## [1] 1.624607e-224
```

Being able to quickly extract key numerical information from a test object is huge. For one thing, it allows you to automate the process of running statistical tests over different datasets or simulations. In Chapter XX, we'll see how you can use loops to do this in a snap.

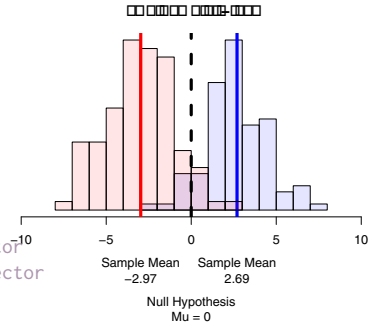
Two-sample t-test

In a two-sample t-test, we use two sets of observations drawn from two different populations and test whether or not the two populations have the same mean. To conduct a two-sample t-test, we simply enter two vectors as arguments `x` and `y`.

Let's use this convention to compare the ages of pirates who wear headbands and pirates who don't wear headbands. First, we'll create the two vectors `age.headband` and `age.noheadband` containing the age data for pirates who do and do not wear headbands. We'll then enter these vectors as arguments `x` and `y` to `t.test()`:

```
age.headband <- subset(pirates, subset = headband == "yes")$age  # Get the first vector
age.noheadband <- subset(pirates, subset = headband == "no")$age  # Get the second vector

test.result <- t.test(x = age.headband, # Enter the first vector
                      y = age.noheadband, # Enter the second vector
                      alternative = "two.sided" # Specify a two-tailed test
                      )
test.result

##
##  Welch Two Sample t-test
##
## data:  age.headband and age.noheadband
## 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:
##  -2.0416673  0.3834475
## sample estimates:
## mean of x mean of y
##  27.31375  28.14286
```

```
par(mar = c(10, 0, 3, 1))
plot(1, xlim = c(-10, 10), ylim = c(0, 1),
     yaxt = "n", main = "Two Sample t-test",
     type = "n", bty = "n", ylab = "", xlab = "")

samples.1 <- rnorm(100, mean = 3, sd = 2)

par(new = T)
hist(samples.1, yaxt = "n", xaxt = "n", xlab = "",
     ylab = "", main = "", xlim = c(-10, 10),
     col = rgb(0, 0, 1, alpha = .1))

mtext(paste("Sample Mean\n", round(mean(samples.1), 2), sep = ""),
      side = 1, line = 3.5, at = mean(samples.1))

abline(v = mean(samples.1), lty = 1,
       col = rgb(0, 0, 1, alpha = 1), lwd = 4)

samples.2 <- rnorm(100, mean = -3, sd = 2)

par(new = T)
hist(samples.2, yaxt = "n", xaxt = "n", xlab = "",
     ylab = "", main = "", xlim = c(-10, 10),
     col = rgb(1, 0, 0, alpha = .1))

mtext(paste("Sample Mean\n", round(mean(samples.2), 2), sep = ""),
      side = 1, line = 3.5, at = mean(samples.2))

abline(v = mean(samples.2), lty = 1,
       col = rgb(1, 0, 0, alpha = 1), lwd = 4)

mtext("Null Hypothesis\nMu = 0", side = 1, line = 6, at =0)

abline(v = 0, lty = 2, col = "black", lwd = 4)
```



Looks like we see a test statistic of -1.35 with a resulting p-value of 0.18. According to null-hypothesis test logic, we fail reject the null hypothesis.

Specifying t-tests with formula notation

The second (and I think better) way to specify arguments to the `t.test()` function is by using the `formula` and `subset` arguments. Using this notation, we specify the dependent and independent variables as a formula in the form `dv ~ iv` where `dv` is the dependent variable, and `iv` is the independent variable with two levels in a dataframe. As you'll see, this convention is a bit nicer to use when working with data in dataframes because we don't need to define two separate vectors prior to the test.

`t.test(formula, data)`

`formula`

An (optional) formula in the form `dv ~ iv` where `dv` is the dependent variable, and `iv` is the independent variable with two levels in a dataframe. Specify the dataframe in the `data` argument

`data`

The dataframe containing the columns specified in `formula`.

`subset`

A logical vector indicating a subset of data to use. If the independent variable in the formula specification has more than two values, you'll need to use `subset` to restrict the data to only two values of the iv.

`alternative, mu, paired, var.equal`

Additional arguments (see previous `t.test()` description)

Using this formulation, we don't have to define separate vectors (x and y) prior to conducting the test. Instead, we can use the `formula` argument to tell R which columns in a dataframe correspond to the dependent and independent variables. When you use the formula version of t.test, the independent variable *must* only have two possible values. If R finds more than two values in the iv, it will return an error. To ensure that there are only two values present, include the appropriate `subset` argument. For example, if the independent

variable is sex and you want to compare males and females, include the argument `subset = sex %in% (c("male", "female"))`

Let's repeat the previous t-test using a formula. To do this, we'll specify three new arguments:

- `data = pirates`: The columns in `formula` come from the dataframe `pirates`
- formula = age ~ headband: Conduct a test on age as a function of headband.
- `subset = headband %in% c("yes", "no")`: Restrict our analysis to pirates who answered `"yes"` or `"no"` to whether or not they wear a headband.

Here's how the alternative notation for the same test looks:

```
test.result <- t.test(formula = age ~ headband,  #dv is weight, iv is Diet
                      subset = headband %in% c("yes", "no"),  # Only use valid headband values
                      data = pirates, # Dataframe is pirates
                      alternative = "two.sided" # Two-sided test
                      )
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
```

As you can see, the results of this test and the prior test are identical. You can use which ever version makes more sense to you. Personally, I like the formula version because all the necessary commands (including the specification that the two diets are 1 and 2) are contained within the `t.test()` function. Of course, you can also specify additional restrictions in the `subset` argument.

Let's try making the previous test a little more complicated by adding a `subset` argument. Let's say a pirate tells you "Oh, well there's only a difference between the age of headband and no-headband pirates for those pirates who went to college at Captain

Chunk's Canon Crew (aka CCCC)." We can test this by adding the additional restriction `college == "CCCC"`, to the `subset` argument:

```
test.result <- t.test(formula = age ~ headband,
                      subset = headband %in% c("yes", "no") &
                               college == "CCCC",
                      data = pirates,
                      alternative = "two.sided"
                      )
test.result

##
##  Welch Two Sample t-test
##
## data:  age by headband
## t = 0.8086, df = 80.352, p-value = 0.4211
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.560535  1.327896
## sample estimates:
##  mean in group no mean in group yes
##          24.82812          24.44444
```

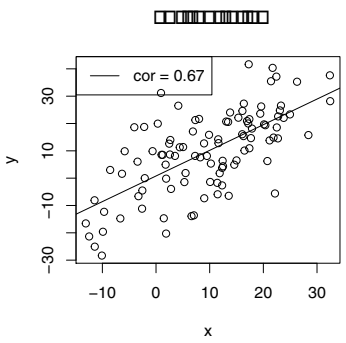
Looks like we still don't find a significant difference in age between headband and no-headband wearers, even just for pirates who went to Captain Chunk's Canon Crew.

Correlation test

Next we'll cover two-sample correlation tests. Recall that in a correlation test, you are accessing the relationship between two variables on a ratio or interval scale.

To run a correlation test, use the `cor.test(x, y)` function. The test has the following arguments

```
x <- rnorm(n = 100, mean = 10, sd = 10)
y <- x + rnorm(n = 100, mean = 0, sd = 10)

plot(x, y, main = "cor.test(x, y)")
abline(lm(y ~ x))

legend("topleft",
       legend = paste("cor = ", round(cor(x, y), 2), sep = ""),
       lty = 1)
```



`cor.test(x, y)`: Conduct a correlation test between two vectors x and y.

cor.test()

`formula OR x, y`

The formula notation for cor.test() is ~ x + y. For example, to conduct a correlation test between height and weight, you'd enter `formula = ~ height + weight`. If your data are in two separate vectors, you can use the vector notation with the arguments x and y.

`alternative`

A string indicating the direction of the test. You can use "t" for two-sided, "l", for less than, and "g" for greater than.

`method`

A string indicating which correlation coefficient to test. You can use "pearson", "kendall", or "spearman". The default is Pearson.

`conf.level`

The confidence level for the Pearson correlation coefficient.

Let's conduct a correlation test on the age of pirates and the number of parrots they've had in their lifetime. We'll set x = pirates$age, and `y = pirates$parrots.lifetime`

```
test.result <- cor.test(x = pirates$age,
                        y = pirates$parrots
                        )

test.result

##
##  Pearson's product-moment correlation
##
## data:  pirates$age and pirates$parrots
## t = 7.4153, df = 998, p-value = 2.589e-13
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.1689151 0.2864504
## sample estimates:
##       cor
## 0.2285152
```

Alternatively, we could use the formula notation. When doing a correlation test, the formula notation is a bit different from what were used to – instead of `formula = y ~ x`, you enter `~ x + y`. We should get the same result as before:

I'm not certain why R uses the `~ x + y` formula notation for `cor.test()` compared to the `y ~ x` notation for `t.test()`. My best guess is that has to do the fact that, while the order of variables matters in a `t.test()`, it does not matter in a correlation test with `cor.test()`. For this reason R may prefer to *not* use the `y ~ x` notation for correlation tests because the `y ~ x` notation makes it look like the order of the variables matters (which, again, it doesn't for a correlation test).

```
test.result <- cor.test(~ age + parrots,
                        data = pirates
                        )

test.result

##
##  Pearson's product-moment correlation
##
## data:  age and parrots
## t = 7.4153, df = 998, p-value = 2.589e-13
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.1689151 0.2864504
## sample estimates:
##       cor
## 0.2285152
```

Looks like we have a positive correlation of 0.23! To see what information we can extract for this test, let's run the command `names()` on the test object:

```
names(test.result)

## [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:

```
test.result$conf.int

## [1] 0.1689151 0.2864504
## attr(,"conf.level")
## [1] 0.95
```

You'll notice that when we tried to access the confidence interval, we got an additional piece of information called `attr(,"conf.level")`. This means that the result of the command `test.result$conf.int` not only contains the bounds of the confidence interval, but also the level of confidence. This is a good thing because the confidence interval only makes sense in terms of the level of confidence used to calculate the interval.

Chi-square test

Next, we'll cover chi-square tests. In a chi-square test test, we test whether or not there is a relationship between two variables on a nominal scale (like sex, eye color, first name etc.). To conduct a chi-square test, we use the `chi.square()` function.

chisq.test()

`x, y`

Two vectors (can be numeric, factor, or string) of the same length. Alternatively, you can simply enter a matrix as the x argument and ignore the y argument.

`correct`

a logical indicating whether to apply continuity correction when computing the test statistic for 2 by 2 tables: one half is subtracted from all |O - E| differences; however, the correction will not be bigger than the differences themselves.

Let's use the `chisq.test()` function to test if there is a relationship between the college a pirate went to and the type of sword he/she uses. We'll use the tattoo and college vectors in `pirates`

Unlike `t.test()` and `cor.test()`, `chisq.test()` does not allow you to use the formula notation. Instead, you have to specify two vectors x and y explicitly.

```
test.result <- chisq.test(x = pirates$college,
                          y = pirates$sword.type)

test.result

##
##  Pearson's Chi-squared test
##
## data:  pirates$college and pirates$sword.type
## X-squared = 2.1502, df = 3, p-value = 0.5418
```

Looks like we got a test-statistic of 2.15 and a p-value of 0.54. Because the p-value is less than .05, we *do* have sufficient data to reject the null hypothesis and conclude that there is a relationship between the two variables.

Let's see what other information we can get from the chi-squre test object:

I encourage you to run the `names()` function on statistical objects. You never know what interesting things you'll discover!

```
names(test.result)

## [1] "statistic" "parameter" "p.value"   "method"    "data.name" "observed"
## [7] "expected"  "residuals" "stdres"
```

We've got some interesting new options here. Let's look at the value of `observed`, the observed frequencies in the data

```
test.result$observed

##                pirates$sword.type
## pirates$college banana cutlass sabre scimitar
##           CCCC      28     548    44       47
##           JSSFP     16     282    16       19
```

Cool. It looks like R stores a table of the observed frequencies and the expected frequencies under the null-hypothesis. Thanks R!

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:

```
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:

```
apa(with(pirates, cor.test(age, parrots)))

## [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.

Additional Tips

Test your R might!

1. Do male pirates have significantly longer beards than female pirates? Test this by conducting the appropriate test on the relevant data in the pirates dataset.
2. Are pirates whose favorite pixar movie is Up more or less likely to wear an eye patch than those whose favorite pixar movie is Inside Out? Test this by conducting the appropriate test on the relevant data in the pirates dataset.
3. Do longer movies have significantly higher budgets than shorter movies? Answer this question by conducting the appropriate test in the movies dataset.
4. Do R rated movies earn significantly more money than PG-13 movies? Test this by conducting a the appropriate test on the relevant data in the movies dataset.
5. Are certain movie genres significantly more common than others in the movies dataset?

12: Regression and ANOVA

Pirates like shiny new things. But getting shiny new things requires gold, and gold doesn't grow on trees (obviously, it is found in treasure chests). The fastest way to get a new batch of gold is to sell an old ship - but how much money can you expect to get from a ship? For example, if you have a 2 year old, Red, classic ship with 8 cannons, 3 rooms, with a weight of 5000kg, how much money could you expect to get for it? To answer this, we'd like to know how the attributes of the ship (e.g.; number of cannons, color) relate to its value. We can get these values using linear regression.

Figure 43: Buy me old ship! It be great for Hipirates (hipster pirates)

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 define the value of a diamond as a linear function of the diamond's weight (x_1) and clarity (x_2). We can then use the linear model to estimate the beta weights (β_1, β_2) that correspond to each independent variable.

To estimate the beta weights of a linear model in R, we use the `lm()` function:

lm()

`function`

A function 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.

`subset`

An optional vector specifying a subset of observations to be used in the fitting process. For example the argument `subset = age > 50 & sex == "male"` will restrict the analysis to points where age is greater than 50 and sex is equal to male.

Let's try an example with the results of a recent pirate ship auction where 1,000 ships were sold to the highest bidder. You can download the dataset at http://nathanieldphillips.com/wp-content/uploads/2015/06/shipauction.txt using the following code chunk:

```
shipauction <- read.table("http://nathanieldphillips.com/wp-content/uploads/2015/06/shipauction.txt",
                          sep = "\t",
                          comment = "")
```

This dataset contains three columns: `cannons`: the number of cannons on the ship, `rooms`: the number of rooms on the ship, `age`: the age of the ship in years, `style`: the style of the ship, `condition`: the condition of the ship on a scale of 1 to 10, `weight`: the weight of the ship, and `price`: The price that the ship sold at.

Before we get started, let's get a look at the dataset by executing `summary` on the dataset

```
summary(shipauction)

##     cannons          rooms            age            style
##  Min.   : 2.00   Min.   :10.00   Min.   :15.00   classic:500
##  1st Qu.: 6.00   1st Qu.:22.00   1st Qu.:43.80   modern :500
##  Median :10.00   Median :34.00   Median :49.80
##  Mean   :10.97   Mean   :34.21   Mean   :50.04
##  3rd Qu.:16.00   3rd Qu.:46.00   3rd Qu.:56.40
```

```
##  Max.   :20.00   Max.   :58.00   Max.   :82.80
##    condition          weight       color         price
##  Min.   : 1.000   Min.   :3481   black:502   Min.   :-23696
##  1st Qu.: 4.000   1st Qu.:4704   brown:286   1st Qu.:  2058
##  Median : 6.000   Median :5058   red  :212   Median : 18458
##  Mean   : 5.622   Mean   :5021               Mean   : 17223
##  3rd Qu.: 7.000   3rd Qu.:5341               3rd Qu.: 32244
##  Max.   :10.000   Max.   :6372               Max.   : 54231
```

Now, let's use the linear model function `lm()` to regress the dependent variable `price` on the independent variables. This will tell us how relevant each of the independent variables was in the final selling price of the ship. We'll run execute the `lm()` command and assign the result to an object called auction.lm:

```
auction.lm <- lm(price ~ cannons + rooms + age +
                   style + condition + weight,
                 data = shipauction)
```

When we call the `lm()` function and assign it to an object, R will save all the results to that object. To see the actual statistical results, we need to run the `summary` command on the test object `auction.lm`. This will output a summary table with estimates of the beta weights as well as statistical tests testing whether the weights are significantly different from 0 - here is the output from the summary function applied to the linear model object `auction.lm` that we just defined:

```
summary(auction.lm)

##
## Call:
## lm(formula = price ~ ., data = shipauction)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -46329  -9988   -510  10425  45766
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)   9877.492   6374.775   1.549    0.122
## cannons        121.381     85.044   1.427    0.154
## rooms          249.987     32.838   7.613 6.24e-14 ***
## age            -77.435     55.438  -1.397    0.163
## stylemodern -16137.521   1055.482 -15.289  < 2e-16 ***
## condition      -69.391    223.372  -0.311    0.756
```

Because we are including all columns in the `shipauction` dataframe as independent variables, we can make the code a bit simpler by replacing all the independent variable names by a period as follows:

```
auction.lm <- lm(price ~ .,
                 data = shipauction)
```

Using this format will save you a lot of typing if you are running regression analyses on dataframes with many independent variables!

```
## weight           1.319      1.037   1.272    0.204
## colorbrown    6560.008   1154.160   5.684 1.73e-08 ***
## colorred      6113.353   1282.337   4.767 2.15e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15570 on 991 degrees of freedom
## Multiple R-squared:  0.2603,Adjusted R-squared:  0.2544
## F-statistic:  43.6 on 8 and 991 DF,  p-value: < 2.2e-16
```

As you can see above, executing the `summary()` function on a linear model object returns several results. The first output is `call` which tells us the exact model we ran. The second output is `Residuals`, which gives us information about the distribution of residuals - the difference between the model fits and the actual response data. In this case, we see that the median residual is -510 which means that the median deviation between the data and the model fits across all data points is -510. The third, and most important, output is `coefficients`, which shows us the estimated beta values for each independent variable.

Let's look at the table of coefficients: for each independent variable (including the intercept β_0), we see an estimate (indicating the estimated beta weight), the standard error of the beta weight, a test statistic testing whether the beta weight is significantly different from 0, and a p-value (which R calls Pr(>|t|))[4] telling us the probability of obtaining a test statistic as large as the one we found assuming that the true population beta weight is truly 0. In the next column, R will output 0, 1, or more stars indicating how small the p-value is. P-values greater than .05 have no stars, indicating 'no significance' at the .05 level, 1 star indicating a p-value less than .05, 2 stars indicating a p-value less than .01 (and so on). Generally, one or more stars indicates that an effect is significant at the .05 level.

[4] Note: The text `Pr(>|t|)` is the p-value.

We can get lots of other information from the linear model object. Here are three of them (to see all of them, run `names(auction.lm)`):

- `coefficients`: A vector of the estimated beta values corresponding to the independent variables (and the intercept)
- `fitted.values`: A vector of the fitted values for all test cases. This is the fitted linear model's prediction for the dependent variable (in our case, price) for each ship.
- `residuals`: A vector of the differences between the true response values and the fitted response values.

These attributes let us easily calculate some interesting model

diagnostics. For example, the attribute `fitted.values` shows us, for each row in the original dataset, what the model predicted the value should be. We can use this information to create a scatterplot comparing the true value for each ship, and the value fitted by the model. If the model did a good job of capturing the data, we would expect a strong linear relationship between the two. See margin figure XXX for an example:

```
plot(x = shipauction$price,
     y = auction.lm$fitted.values,
     xlab = "True Price of Ship",
     ylab = "Model Price of Ship",
     main = "Pirate ship auction prices\n True versu
     pch = 16, col = gray(.05, .15)
     )

# Add diagonal line
abline(a = 0, b = 1)
```

To get a full analysis of variance table (ANOVA) from a linear regression object, we can use the `anova()` function. The output of this function will look very similar to the table we saw after running `summary()`, but it contains a bit more information

```
anova(auction.lm)

## Analysis of Variance Table
##
## Response: price
##            Df     Sum Sq    Mean Sq  F value    Pr(>F)
## cannons     1 6.6868e+08 6.6868e+08   2.7597   0.09698 .
## rooms       1 1.2957e+10 1.2957e+10  53.4758 5.392e-13 ***
## age         1 4.6067e+09 4.6067e+09  19.0125 1.434e-05 ***
## style       1 5.5730e+10 5.5730e+10 230.0058 < 2.2e-16 ***
## condition   1 1.0228e+07 1.0228e+07   0.0422   0.83725
## weight      1 4.0155e+08 4.0155e+08   1.6573   0.19827
## color       2 1.0142e+10 5.0708e+09  20.9279 1.253e-09 ***
## Residuals 991 2.4012e+11 2.4230e+08
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

If you want to access the results of an anova table directly, you can use the $ notation. Let's use the `names()` function to see what is in the anova object:

```
names(anova(auction.lm))

## [1] "Df"      "Sum Sq"  "Mean Sq" "F value" "Pr(>F)"
```

Now we can use these names to access specific columns from the anova table. For example, to get the p-values, we can run the following:

```
anova(auction.lm)$"Pr(>F)"

## [1] 9.697992e-02 5.392042e-13 1.434304e-05 7.084165e-47 8.372540e-01
## [6] 1.982739e-01 1.252630e-09           NA
```

This vector of p-values is given in the same order of the independent variables in the previous ANOVA table. For example, the first value is the p-value for `cannons`, and the second p-value is for `rooms`.

5

[5] I know what you're thinking "Why did we put "Pr(>F)" in quotation marks?" The reason is because R gets confused by the > symbol in the column name. We use quotation marks to tell R that Pr(>F) is just a name and is not meant to be a mathematical operation. It's a strange quirk in R, I wish they had just called the p-value column p or something, but that's how it is....

*Interactions in model terms: y ~ x1 * x2*

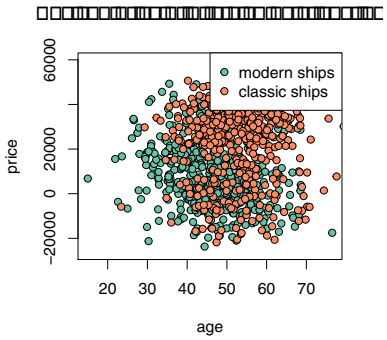
An interaction in a linear model means that the relationship between one independent variable on the dependent variable depends on the level of another independent variable. For example, let's see if the relationship between age and price depends on the level of style. To keep things simple, we'll only look at this one interaction (and ignore other independent variables). To include interaction terms in a linear model, use the $x1 \times x2$ notation. Because we're testing the interaction between age and style, we'll enter `cannons * style` as the independent variable(s):

```
auction.lm <- lm(price ~ age * style,
                 data = shipauction)
summary(auction.lm)

##
## Call:
## lm(formula = price ~ age * style, data = shipauction)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -47312 -10939   1599  11210  38236
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)     14122.20    4491.63   3.144  0.00172 **
## age               202.23      82.96   2.438  0.01495 *
## stylemodern      9162.84    5864.24   1.562  0.11849
## age:stylemodern  -497.37     114.78  -4.333 1.62e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 16160 on 996 degrees of freedom
## Multiple R-squared:  0.1988,Adjusted R-squared:  0.1964
## F-statistic:  82.4 on 3 and 996 DF,  p-value: < 2.2e-16
```

As you can see from the summary table, we have both significant main effects for both independent variables, and a significant main effect for the interaction (labelled `age:stylemodern`). The estimate for the interaction term (in this case, -497.37) tells us that, for modern ships, the beta weight for age is estimated to be -497.37) smaller than the beta weight for cannons for classic ships. To see this relation-

ship, let's create two scatterplots showing the relationship between age and price, one for modern ships and one for classic ships. If we interpreted the interaction correctly, we should see a smaller slope between age and price for modern ships than for classic ships. The two scatterplots are shown in margin figure XXX:

```
require(RColorBrewer)
col.vec <- brewer.pal(3, "Set2")
modern.data <- subset(shipauction, style == "modern")
classic.data <- subset(shipauction, style == "classic")

plot(modern.data$age, modern.data$price, pch = 21,
     bg = col.vec[1],
     ylim = c(min(shipauction$price) * 1.1, max(shipauction$price) * 1.1),
     xlab = "age",
     ylab = "price",
     main = "Age and price of pirate ships at auction"
     )

points(classic.data$age, classic.data$price, pch = 21, bg = col.vec[2])

legend("topright", legend = c("modern ships", "classic ships"),
       pch = c(21, 21), pt.bg = col.vec, bg = "white")
```



Looking at the graph, we can see that our previous conclusion make sense: for modern ships, we see a negative relationship between age and price, the older a modern ship is, the smaller the price it receives. However, for classic ships, we actually see a positive relationship between age and price: the older a classic ship is, the higher the price it receives.

Calculating an ANOVA with `aov()`

Let's go over how to calculate an ANOVA using the `aov()` function. You would calculate an ANOVA to test whether there is a relationship between a nominal independent variable and a quantitative dependent variable. For example, is there a relationship between the color of a pirate ship and its selling price? Let's test this by using `aov()`. We'll set price as the dependent variable and color as the independent variable.

```
color.aov <- aov(price ~ color, data = shipauction)
summary(color.aov)

##              Df    Sum Sq   Mean Sq F value   Pr(>F)
## color         2 9.319e+09 4.660e+09   14.73 4.94e-07 ***
## Residuals   997 3.153e+11 3.163e+08
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As you can see in the summary table, because the p-value of 0 is less than .05, we conclude that we do find a significant effect of color on ship prices. After finding a significant ANOVA, we can move on to conduct post-hoc tests to test pair-wise differences between the colors. We can do this using the `TukeyHSD()` function:

```
TukeyHSD(color.aov)

##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = price ~ color, data = shipauction)
##
## $color
```

```
##                  diff       lwr      upr     p adj
## brown-black  6580.029  3487.567 9672.492 0.0000021
## red-black    5314.726  1895.676 8733.776 0.0008096
## red-brown   -1265.304 -5048.334 2517.727 0.7123070
```

Looking at the output of this function, we can see statistical tests comparing all pairs of the three colors. The four columns in the table are as follows:

1. diff: The difference in means between the two groups.

2. lwr, upr: The lower and upper bounds of the 95% CI of the difference between the two groups

3. p adj: The adjusted p-value testing whether or not the difference in groups is significantly different from 0. The p-value is adjusted so the family-wise type 1 error rate is 95%. Look at ?TukeyHSD for more information.

For example, let's look at the first row comparing brown to black ships. We find a mean difference of 6580.0294208, a 95% confidence interval of the difference of [3487.57, 9672.49], and an adjusted p-value of 0. Because the p-value is smaller than .05, we conclude that there is a significant difference in the mean value of brown and black ships.

Generalized Linear Model (GLM)

In the Generalized Linear Model (GLM), we take the original linear model, but apply a link function that wraps around the linear combination of predictors. This allows us to model response data that is not normally distributed.

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?"

The key new argument in `glm()` compared to `lm()` is the `family` argument. This argument tells R which link function to use. To see more information about the families, look at help under `?family`.

Binary Logistic Regression

Probably the most common non-Normal family you will use is binomial which corresponds to binary logistic regression. In binary logistic regression, we predict a binary outcome variable (containing 0s and 1s) as the logit transformation of a linear combination of a set of predictors. Formally:

$$p(y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... \beta_n x_n)}}$$

To conduct binary logistic regression, we use the `family = "binomial"` argument. Let's try an example with our `shipauction` dataset. Let's make age our independent variable, and style as our dependent variable. This will allow us to test whether older ships are more likely to be classic ships than modern ships. However, before we run the

```
# Logit
logit.fun <- function(x) {1 / (1 + exp(-x))}

curve(logit.fun,
      from = -3,
      to = 3,
      lwd = 2,
      main = "Logit",
      ylab = "p(y = 1)",
      xlab = expression("b_{0} + b_{1}x_{1} + b_{2}x_{2} + ... b_{n}x_{n}")
      )

abline(h = .5, lty = 2)
abline(v = 0, lty = 1)
```

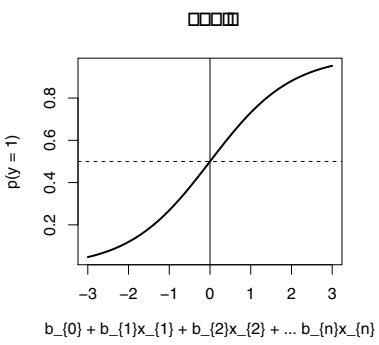


Figure 44: The logit function used in binary logistic regression

analysis, let's recode the style column into a numeric variable, where `modern = 0` and `classic = 1`. This will help us to interpret the beta values in the model later on:

```
shipauction$style.num[shipauction$style == "modern"] <- 0
shipauction$style.num[shipauction$style == "classic"] <- 1
```

Now we're ready to fit the binary logistic regression model. We'll enter our new style.num variable as the dependent variable, and age as the independent variable. Because we want to do binary logistic regression, we'll enter "binomial" as the family:

```
age.style.glm <- glm(style.num ~ age,
             data = shipauction,
             family = "binomial")
```

Now let's look at the results!

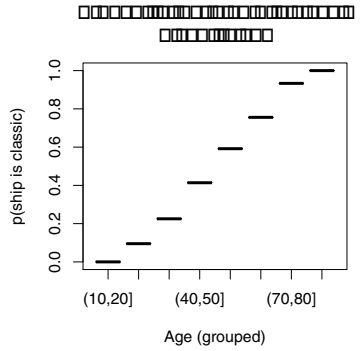
```
summary(age.style.glm)

##
## Call:
## glm(formula = style.num ~ age, family = "binomial", data = shipauction)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.17931  -1.05773   0.01562   1.05759   2.18632
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -4.308187   0.413836  -10.41   <2e-16 ***
## age          0.086081   0.008157   10.55   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1386.3  on 999  degrees of freedom
## Residual deviance: 1250.4  on 998  degrees of freedom
## AIC: 1254.4
##
## Number of Fisher Scoring iterations: 4
```

Looking at the summary table, it looks like we have a significant positive effect of age (beta = 0.086, p = 0), meaning that the older a ship is, the more likely it is that it's a classic ship (we know it's more

likely to be classic because we coded classic ships as 1 and modern ships as 0. If we reversed the coding, the sign of the beta weight would become negative).

You can visualize this relationship in the margin figure. In the figure, I group the ships into 10 groups according to their age using the `cut()` function and then calculated the proportion of ships in each age group that were modern using `aggregate()`:

```
shipauction$style.classic <- shipauction$style == "classic"

shipauction$age.cut <- cut(shipauction$age,
                           breaks = seq(10, 90, 10))

probs <- aggregate(style.classic ~ age.cut,
                   data = shipauction, FUN = mean)

plot(probs, xlab = "Age (grouped)",
     ylab = "p(ship is classic)",
     main = "Probabiliy that a ship is classic\ngiven its age")
```

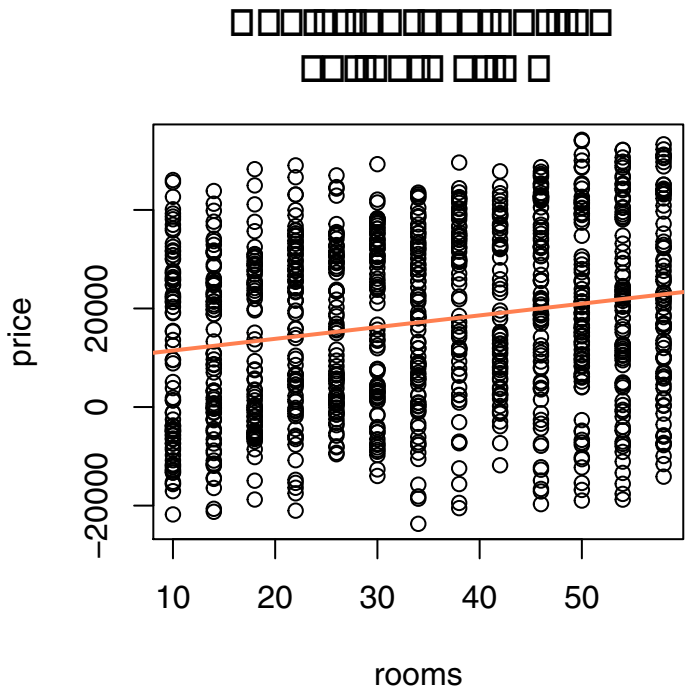


Additional Tips

Adding a regression line to a plot

Once you've generated a model object from `lm()` or `glm()`, you can easily add a regression line from that object to a graph. To do this, use the `abline()` function with the linear model object as the primary argument. For example, to add a regression line to a scatterplot showing the relationship between `rooms` and `price`, we can run the following:

```
plot(x = shipauction$rooms,
     y = shipauction$price,
     xlab = "rooms",
     ylab = "price",
     main = "Add a regression line\nabline(my.lm)")

rooms.price.lm <- lm(price ~ rooms,
                     data = shipauction)

abline(rooms.price.lm,
       lty = 1,
       lwd = 2,
       col = "coral")
```



Using a regression model to predict new data

Once you've saved a model object, you can use the `predict()` function to make predictions for new datasets with the model. For example, let's say you want to predict the selling price of a two pirate ships, the Muskrat and the Dirty Mane, using a regression model. First, let's create the regression model that tries to predict the price of a ship based just on its age and the number of cannons it has. Like in the beginning of this chapter, we'll fit the model using data from the shipauction dataset:

```
auction.lm <- lm(price ~ age + cannons,
                 data = shipauction)
```

Now, we need to create a dataframe representing new data whose price we'd like to predict. Let's create a dataframe called new.ships containing data for our two ships: the Muskrat and the Dirty Mane.

I explicitly assigned the ship names to the rownames of the dataframe so that the names would show up in our final results.

```
new.ships <- data.frame(
                      "ship.name"  = c("Muskrat", "Dirty Mane"),
                       "age" = c(30, 15),
                       "cannons" = c(8, 2)
                       )
```

```
rownames(new.ships) <- new.ships$ship.name
```

Now, we can use the `predict` function to make selling price predictions for our new ships. All we have to do is enter the regression model as the first argument, and our new.ships dataframe as the newdata argument:

```
predict(auction.lm,
        newdata = new.ships)

##    Muskrat Dirty Mane
##  12036.160   7614.341
```

It looks like, according to our regression model, the Muskrat should sell for 1.2036×10^4 and the Dirty Mane should sell for 7614. Of course, these are just estimates, and the true selling price might be very different! To get a true estimate of the uncertainty in the ships' selling price, you'll need to use Bayesian statistics.

13: Writing your own functions

Figure 45: Functions. They're kind of a big deal.

Why would you want to write your own function?

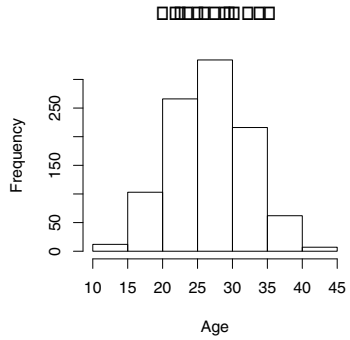
Throughout this book, you have been using tons of functions either built into base-R – like `mean()`, `hist()`, `t.test()`, or written by other people and saved in packages – like `pirateplot()` in the yarrr package, or `chordDiagram()` in the `circlize` package. However, because R is a complete programming 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 (yes, you use functions to create functions! Very Inceptiony), 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.

```
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
```

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
##
## }
```

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...

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)
}
```

You can use any kind of object as an input to a function. For example, we could re-create the function `my.fun` 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.

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.

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

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.

```
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

One very common argument in functions is either a logical or string argument that can tell the function to calculate things in a different way or add additional commands. To do this, we can use the `if()` function in R. The `if()` function has two main elements, a logical test and a chunk of code (in curly braces) that is evaluated only if the logical test is TRUE. If the logical test is FALSE, R will completely ignore all the code in the curly braces.

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"
```

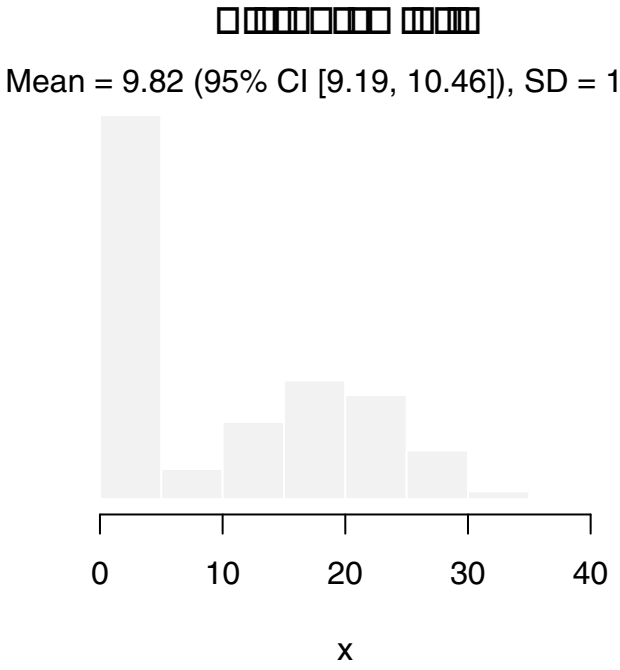
Using `if()` statements in your functions can allow you to do some really neat things. Let's create a function called `show.me()` that takes a vector of data, and either creates a plot, tells the user some statistics, or tells a joke! The function has two inputs: `x` – a vector of data, and `what` – 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[6]. The ... input tells R that the user might add additional inputs that should be used later in the function.

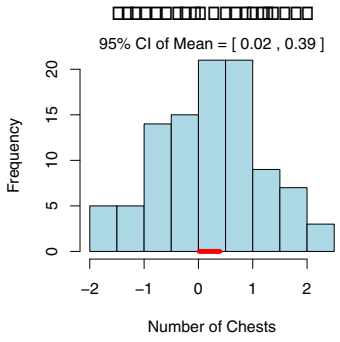
Here's a quick example, let's create a function called `hist.advanced` that plots a histogram with some optional additional arguments passed on with ...

[6] 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

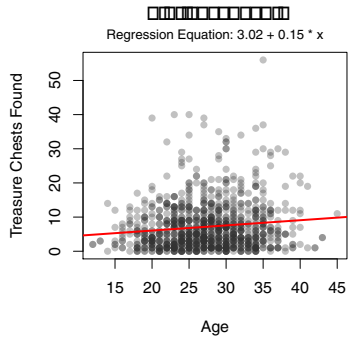
This plotting code is a bit long, but it's all stuff you've learned before.

```
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))
```



14: Loops

Figure 46: 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`. Becaue 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:

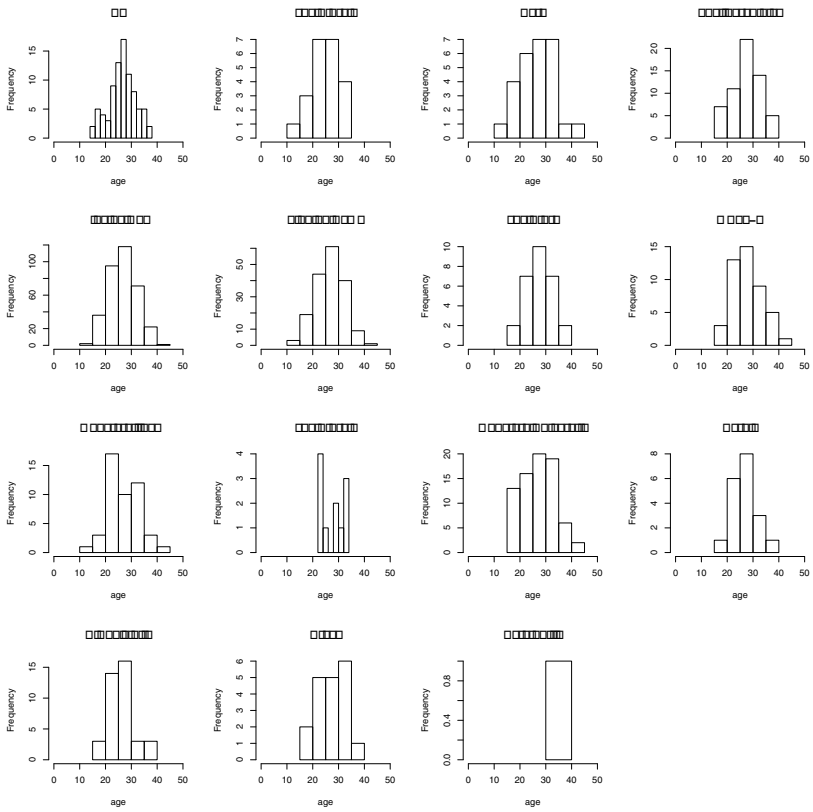
```
100 * 101 / 2

## [1] 5050
```

This boy grew up to be Gauss, a super legit mathematician.

Figure 47: 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 48: 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 1.666667 5.492942 35
## 2   21 2.264706 6.264914 33
## 3   22 2.150000 6.954436 39
## 4   23 2.300000 7.892871 59
## 5   24 1.796875 7.154088 63
## 6   25 2.323529 10.41213 67
## 7   26 2.415094 6.462757 52
## 8   27 2.702381 8.655886 83
## 9   28 2.400000 8.053258 59
## 10  29 2.532258 7.218367 61
## 11  30 2.946667   7.4094 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 31.68750 2.437023
## 2    CCCC   male         24 23.55000 3.833960
## 3   JSSFP female         34 33.67355 3.151243
## 4    CCCC female         27 26.06827 3.520584
```

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]  0.1913454 -1.3449525 -1.3397559  0.6060065  0.1897552 -1.1009423
##  [7] -0.7488270  0.1685509  1.2548874  0.3645940
##
## $second
## [1] -0.8534983  1.2528990 -0.8493408  0.6934078 -1.2732125
##
## $third
```

```
##  [1] -0.94997476  0.31652781 -0.78310862  0.07011897 -0.62728386
##  [6]  1.32791821 -0.82504009 -0.33572765  3.06089665 -1.04049834
## [11]  1.95144059 -0.04863850  1.85433479  0.82003273  0.04018673
```

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]  0.1913454 -1.3449525 -1.3397559  0.6060065  0.1897552 -1.1009423
##  [7] -0.7488270  0.1685509  1.2548874  0.3645940

number.list$second

## [1] -0.8534983  1.2528990 -0.8493408  0.6934078 -1.2732125
```

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] -2.0289
##
## [[2]]
## [1] 0.7209874 0.2263264
##
## [[3]]
## [1] -0.1013256 -0.9588534  0.7809558
##
## [[4]]
## [1]  1.2935419 -1.4292596 -1.4951401 -0.4291774
##
## [[5]]
## [1]  1.0059774 -0.2837458  0.4159475  0.0902826  0.9920353
```

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] -2.0288998  0.7209874  0.2263264 -0.1013256 -0.9588534  0.7809558
##  [7]  1.2935419 -1.4292596 -1.4951401 -0.4291774  1.0059774 -0.2837458
## [13]  0.4159475  0.0902826  0.9920353
```

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[7]. Once you execute this, R will prepare those slaves in the background.

[7] 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 3.2.2 (2015-08-14)

## 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")
```

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()`[8] 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:

[8] 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)
```

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

##    JSSFP     CCCC
## 33.23123 24.48126
```

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,]
```

```
##   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,]

##   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,]

##   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,]

##   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] 9.721739
## 2    (20,30] 9.320000
## 3    (30,40] 9.334532
## 4    (40,50] 8.142857
```

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
     )
```

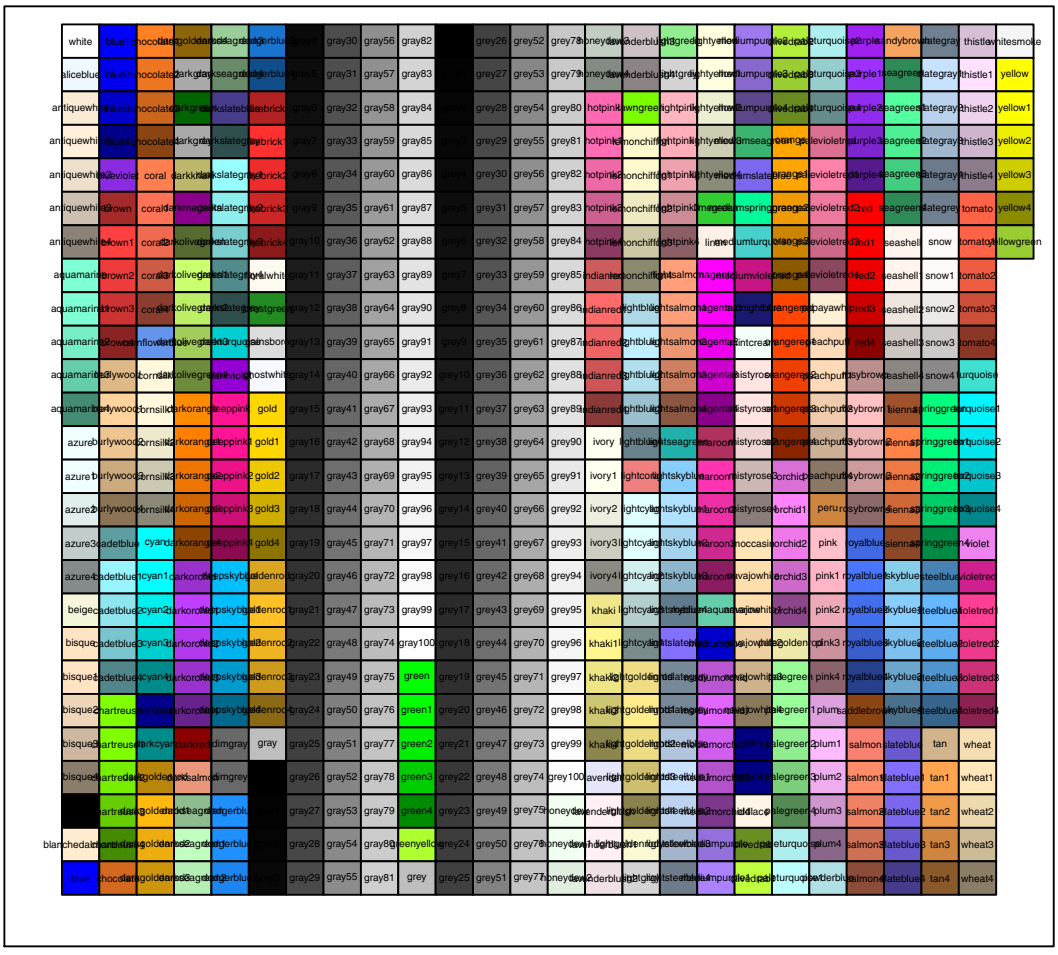


Figure 49: The colors stored in `colors()`.

Index