YaRrr!
The Pirate's Guide to R

DR. NATHANIEL D. PHILLIPS

YARRR! THE PIRATE’S GUIDE TO R

PUBLISHED BY

http://www.nathanieldphillips.com

First printing,

Contents

This book is dedicated to my statistics instructors
Dr. Thomas Moore and Dr. Wei Lin who
taught me everything I know about statistics,
and my PhD colleague Dr. Dirk Wulff
who helped me get started with R.

Introduction

Who am I?

I am a pirate on the Bodensee in Konstanz Germany. When I started pirate training, I discovered R and have been hooked ever since. I'm now on a mission to convince everyone I can to make the switch from SPSS (or Excel, Matlab, JMP...) to R.

This book is in progress..

If you haven't figured it out already, this book is very much a work in progress. 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?

Anyone who wants to learn R can benefit from this book. I will assume that you have taken an introductory course in statistics, but have no substantial programming experience. 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.

```
require("circlize")

## Loading required package:  circlize

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

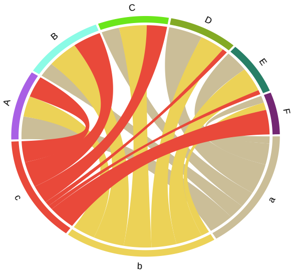


Figure 1: This is a `chordDiagram` plot that comes with the R package `circlize`.

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

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

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

completely transparent!

5. And most importantly of all, R is the programming language of choice for pirates, (who prefer the "YaRrr!" pronunciation)

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

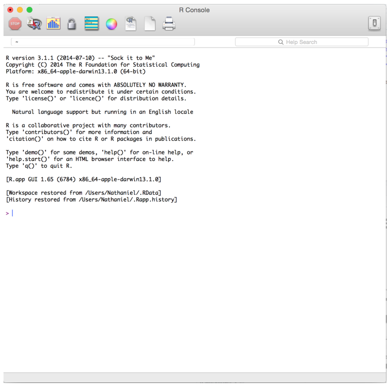
Because this is a beginner's book, I try to avoid bombarding you with too many details and tips when I first introduce a function. For this reason, at the end of every chapter, I present several tips and tricks that I think most R users should know once they get comfortable with the basics. I highly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

1: Installing R, RStudio, and the yarrr package

Now that I've convinced you to use R, let's get started! First, you'll need to install the base R software.

1. Download and install the base R software (around 50mb) + Windows <http://cran.r-project.org/bin/windows/base/> + Mac <http://cran.r-project.org/bin/macosx/>

See Figure 2 Here's how the base R software looks (on Mac). As you can see, it's very much a bare-bones software - just how we want it! No extra gimmicks or flashy bloatware needed!



```
R version 3.1.1 (2014-07-10) -- "Sock it to Me"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.1.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

[R.app GUI 1.65 (6784) x86_64-apple-darwin13.1.0]

[Workspace restored from /Users/Nathaniel/.RData]
[History restored from /Users/Nathaniel/.Rapp.history]

> |
```

Figure 2: Here is how the standard R application looks. Not too exciting - just how we like it!

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. To download and install RStudio (around 40mb), go to <http://www.rstudio.com/products/rstudio/download/>

Once you've installed RStudio, you'll never need to open the base R application. Let's go ahead and boot up RStudio and see how she looks!

The four RStudio windows

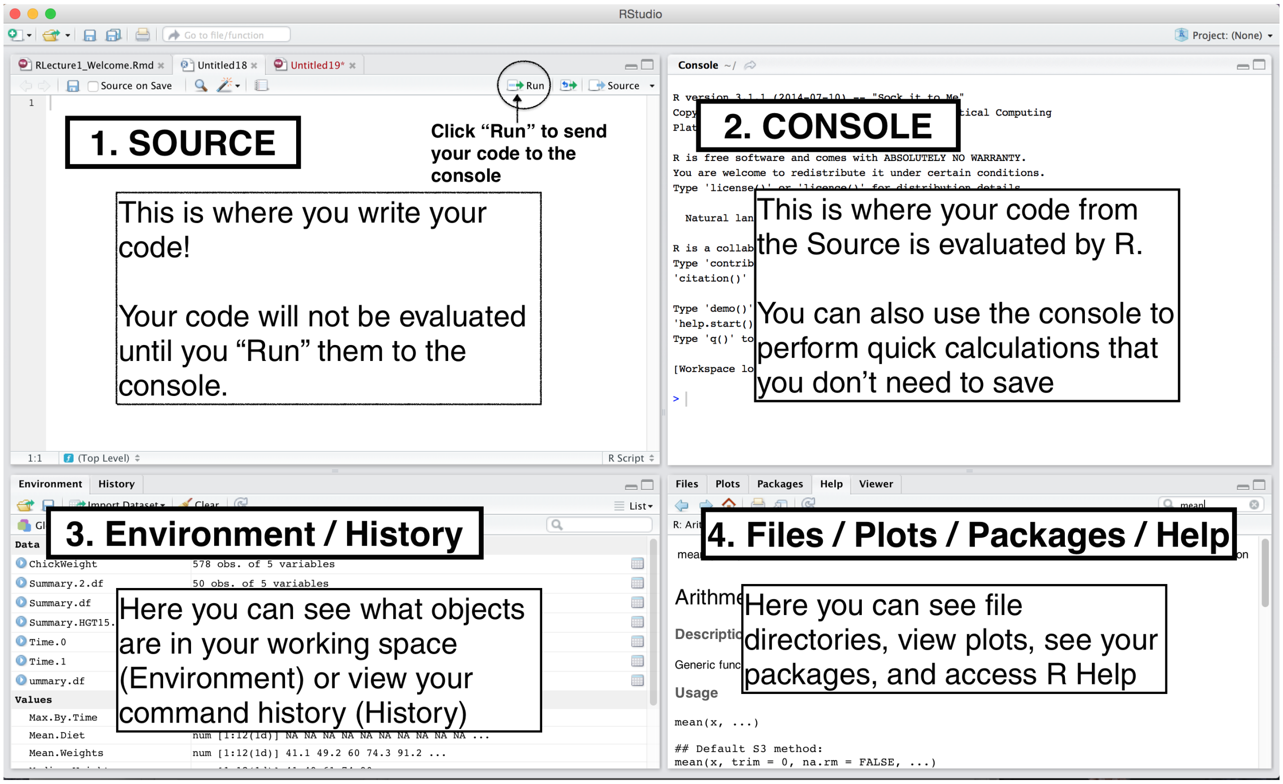
When you open RStudio, you'll see the following four windows (also called panes):

Note your windows might be in a different order. You can change the order of the windows under RStudio preferences.

Source - Your notepad for code

The source pane is where you create and edit R Scripts - which are just text files with the ".R" extension. When you open RStudio, it will automatically start a new Untitled script. You will write 99% of your R code in a script in the source panel. However, your R code will not be evaluated until you 'send' the code to the Console.

You can send your code from the source to the Console by highlighting the code you wish to evaluate and clicking on the "Run" button on the top right of the Source. Alternatively, you can use the



1. SOURCE

Click "Run" to send your code to the console

This is where you write your code!

Your code will not be evaluated until you "Run" them to the console.

2. CONSOLE

This is where your code from the Source is evaluated by R.

You can also use the console to perform quick calculations that you don't need to save

3. Environment / History

Here you can see what objects are in your working space (Environment) or view your command history (History)

4. Files / Plots / Packages / Help

Here you can see file directories, view plots, see your packages, and access R Help

Figure 3: The four panes of RStudio.

hot-key "Command + Return" on Mac, or "Control + Enter" on PC to send code to the console.

Console: The calculator

The console is where R actually executes (calculates) 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
```

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.

However, most of the time, you won't be typing directly into the console. Instead, you'll be writing code in the source and then "Running" it to the console. The reason for this is straightforward: If you type code into the console, it won't be saved (though you can look back on your command History). And if you make a mistake in typing code into the console, you'd have to re-type everything all over again. Instead, it's better to write all your code in the Source. When you are ready to execute some code, you can then send "Run" it to the console.

Environment / History

The Environment tab of this panel shows you the names of all the data objects (like vectors, matrices, and dataframes) that you've defined in your current R session. The tab also has a few clickable actions like importing a new dataset. However, I almost never look at this menu.

The History tab of this panel simply shows you a history of your R commands. I never look at this. In fact, I didn't realize it was even there until I started writing this tutorial.

Files / Plots / Packages / Help

This panel shows you file directories, plots, your current packages, and help menus.

1. Files - 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."

2. Plots - Shows 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.)

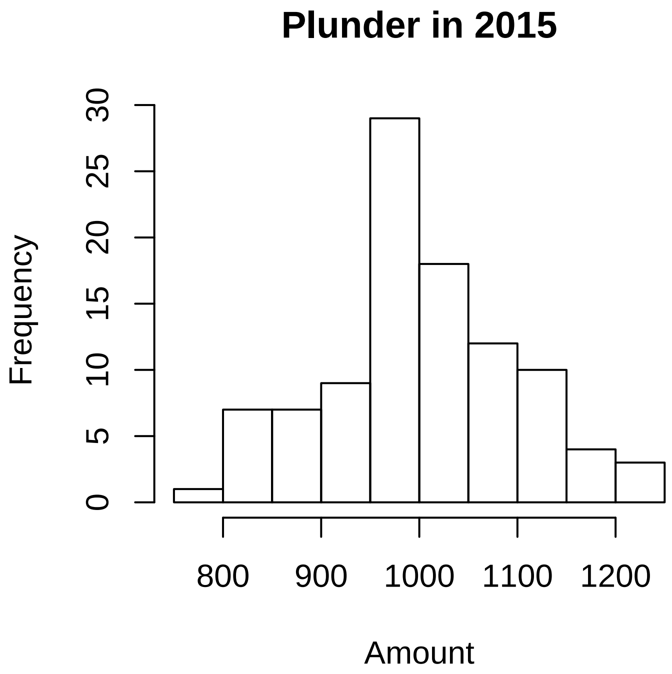
3. Packages - Shows a list of all the R packages installed on your harddrive and indicates whether or not they are currently loaded. Packages that are loaded in the current session are checked while those that are installed but not yet loaded are unchecked.

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`

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.

To see how plots are displayed try the following command which should display a histogram of 100 values randomly drawn from a standard normal distribution.

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



Getting help

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 CRAN (<https://cran.r-project.org/>), the Comprehensive R Archive Network. 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 wordcloud plots.

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(rep(c("Blackbeard", "Jolly.Roger", "Grogg", "Monkey.Island", "Treasure", "cutlass"),
              times = c(100, 20, 50, 40, 200, 100)))

## Loading required package:  tm
  ## Loading required package:  NLP
```

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 `yarr` package on your machine, you'll first need to install the `devtools` package which will then allow you to directly install packages from github:

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

```
install_github('ndphillips/yarrr') # Install the yarrr package from github
library("yarrr") # Load the yarrr package
```

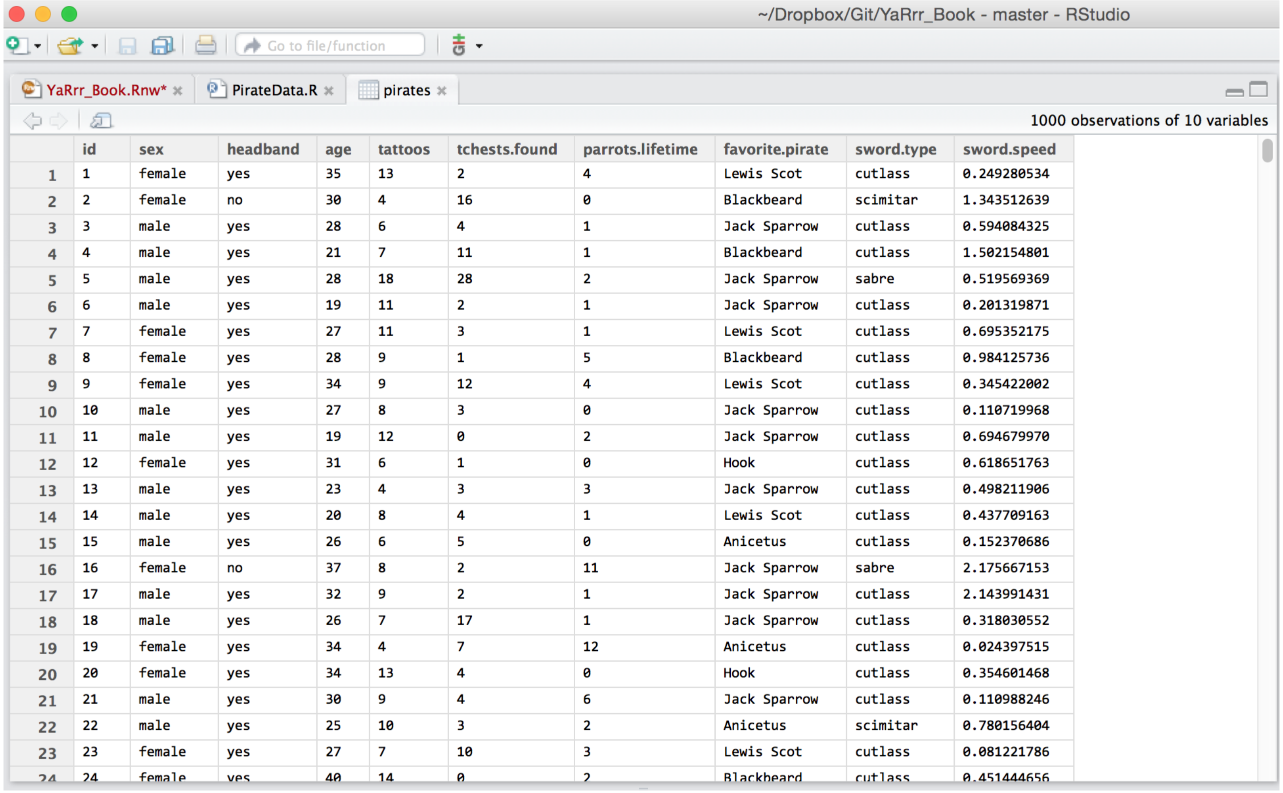
If everything went correctly, you should have access to, among other things, the `pirates` dataset. To see the dataset, you can execute

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

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:)


~/Dropbox/Git/YaRrr_Book - master - RStudio
Go to file/function
YaRrr_Book.Rnw*
PirateData.R
pirates
1000 observations of 10 variables

	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 4: The pirates dataset.

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. You can download a copy at <http://nathanieldphillips.com/wp-content/uploads/2015/09/R-Reference-Card.pdf>. I highly encourage you to print this out and start highlighting functions as you learn them!

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()

Before we get started, a word of warning...

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."

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

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

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.

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:

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.

```
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.

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.

Let's try some more simple calculations

```
2 * 3 - 1 # R ignores spaces

## [1] 5

2 * (3 - 1) # R observes order of operations

## [1] 4
```

As you can see, R ignores spaces in between arguments in code. I recommend using spaces to make your code easier to look at. Personally, I include spaces between arithmetic operators (like + and -) and after commas (which we'll get to later).

Defining 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. Let's start by creating the object `a` and assigning the outcome of $5 * 10$ to it:

Good object names strike a balance between being easy to type (i.e.; short names) and interpret. If you have several datasets, it's probably not a good idea to name them `a`, `b`, `c` because you'll forget which is which. However, using long names like `March2015Group1OnlyFemales` and `March2015Group1OnlyMales` will give you carpel tunnel syndrome.

```
# The symbol(s) "<-" mean "assign"
a <- 5 * 10 # Assign the value of 5*10 to a new object called a
```

Now, anytime we want to refer to the content of the object `a`, we can just type it. When you assign a value to an object, R won't automatically print it. If you want to see the value, you need to call the object by just executing its name:

```
a # What is the value of a?

## [1] 50
```

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

Here are some examples of *invalid* object names:

```
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
```

R is case-sensitive. If you define an object with uppercase letters, you must keep referring to it with uppercase letters!

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.

```
Plunder <- 1
plunder <- 100
Plunder

## [1] 1

plunder

## [1] 100
```

Once you've defined an object, you can use it in other commands. Let's define two objects `a` and `b`, then do other operations on those objects.

```
a <- 100
b <- 2
c <- a + b
c

## [1] 102
```

If you want to change an object, you need to assign it again to the new result you want. If you want to, you can even refer to the same object while assigning it. For example, let's say you have an object `a` with a value of 10 and you want to increase the value of `a` by 10 so the new value of a will be 20.

```
a <- 10   # assign a value of 10 to the object a
a <- a + 10  # assign a to be the object a plus 10
a # Print the new value of a (should be 20)

## [1] 20
```

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

Important!!

It is important to note that once you've defined an object, you refer to it without quotation marks, even if the object is a character. For example, to refer to the object a that I defined above, you need to write a without quotations marks

```
a # Print the value of the object ship

## [1] 20
```

If you write "a" with quotation marks, R will think that you're referring to a new string object

```
"a"   # Just the letter "a", not the object a

## [1] "a"
```

Data object types in R

There are different types of objects in R. The first two objects we'll learn about are scalars and vectors. Later on, we'll talk about more complicated objects like matrices, dataframes, hypothesis tests, etc.

Two simple data objects: scalars and vectors

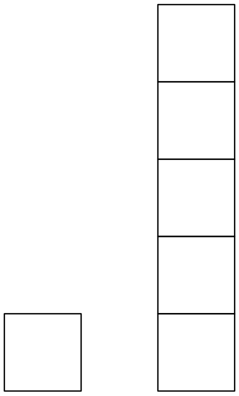
Two of the most common data objects in R are **scalars** and **vectors**. Let's discuss each in turn,

scalars

A **scalar** is just a single value. A scalar can either be *numeric* or *string*. A numeric scalar is a number, while a string scalar is a letter. We denote string scalars by using quotation marks. Here are some examples:

```
a <- 1
ship <- "Black Pearl"
```

```
# 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")
```



Vectors

A **vector** is a combination of several scalars. For example, the numbers from one to ten could be a vector of length 10. Like scalars, you can also create character vectors that contain character scalars.

There are many ways to create vectors in R, here are the most common:

```
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:

The following code will create a vector of the integers from 1 to 5:

`c(x, y, z)`: Create a vector with the c() command by separating elements with commas

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

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

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

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

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

You can also create string vectors containing only string values:

```
a <- c("this", "is", "a", "string", "vector")
a

## [1] "this"   "is"     "a"      "string" "vector"
```

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"
```

Once you've created a vector, you can easily determine its length by using the `length()` function:

`length()`

```
length(c(1, 2, 3))

## [1] 3
```

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`

`a`

The start of the sequence

`b`

The end of the sequence

The `a:b` function creates a vector of numbers from the starting point a to the ending point b in steps of 1:

```
1 : 10 # Integers from 1 to 10

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

10 : 1 # Integers from 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)`

`from`

The start of the sequence

`to`

The end of the sequence

`by`

The step-size of the sequence

`length.out`

The desired length of the final sequence (only use if you don't specify `by`)

The `seq()` function allows you to create a sequence from a starting number to an ending number, in steps you specify. The function has three arguments, which are inputs to the function which changes how it works.

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`

```
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
```

The `rep()` function rep allows you to repeat a number (or vector) a specified number of times. There are three arguments to the rep function:

`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

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.

```
rep(1:5, times = 2) # Repeat integers 1 to 5 two times

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

rep(1:5, each = 2) # Repeat each integer from 1 to 5 two times

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

rep(1:5, each = 2, times = 2) # Do both!

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

Arithmetic operations on scalars and vectors

You can do basic arithmetic operations like +, -, * and / on scalars and vectors. If you do an operation on a vector with a scalar, R will apply the scalar to each element in the vector:

```
a <- 1:5
a * 10

## [1] 10 20 30 40 50

a - 1

## [1] 0 1 2 3 4

a ^ 2

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

If you do an operation on two vectors, R will try to apply the operation between the vectors by each item:

```
1:10 + 21:30

##  [1] 22 24 26 28 30 32 34 36 38 40

(1:5) * (1:5)
```

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

seq(10, 100, 10) + 1:10

##  [1]  11  22  33  44  55  66  77  88  99 110
```

Additional Tips

1. If you want to run a single line of code, you don't need to highlight anything. If you run code (either by clicking the Run button or using the `command + enter` hotkey combination, R will execute the line of code your cursor is on - and automatically move the cursor to the next line. If you want slowly go through several lines of code to see what each line does, I highly recommend using this technique!

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

3. For great blog articles on R, check out http://www.r-bloggers.com/

4. 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.

5. 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 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: Sampling data and Descriptive Statis

Chapter Goals

1. Learn functions for generating data from probability distributions: rnorm(), runif(), sample()

2. Learn functions for basic descriptive statistics: mean(), median(), sd(), var(), min(), max()

Sampling data from probability distributions

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

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 6 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

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

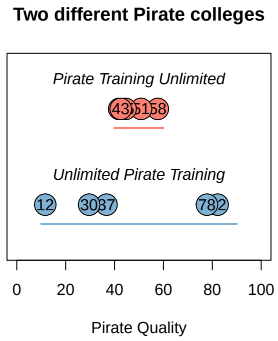


Figure 6: Sampling 5 potential pirates from two different pirate colleges. Pirate Training Unlimited (PTU) consistently produces average pirates (with scores between 40 and 60), while Unlimited Pirate Training (UPT), produces a wide range of pirates from 0 to 100.

The Normal (Gaussian) distribution

Let's start with the most famous distribution in statistics: the Normal (or if you want to sound pretentious, the Gaussian) distribution. The Normal distribution is bell-shaped, and has two parameters: a mean and a standard deviation. See the margin figure 7 for plots of three different Normal distributions with different means and standard deviations.

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

`rnorm()`

`n`

The number of observations

`mean`

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

`sd`

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

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

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

## [1]  0.005568711  0.635282822  0.644565417 -0.028134767  0.194183418
```

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:

```
a <- rnorm(5, mean = 10, sd = 5)
b <- rnorm(5, mean = 10, sd = 5)
a # print a
```

```
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]
       )
```

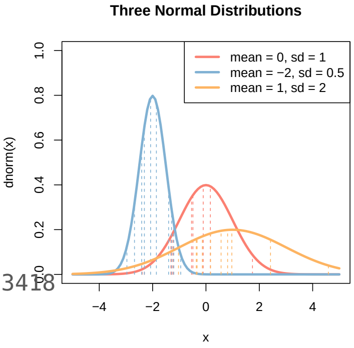


Figure 7: Three different normal distributions with different means and standard deviations.

```
## [1] 10.487477 23.472031 18.582602  7.393501 15.688765

b # print b

## [1] 20.749317  4.571014  8.565096  2.984800  6.382865
```

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 7 on your machine several times. You should see the sampling lines dance around the distribution.

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

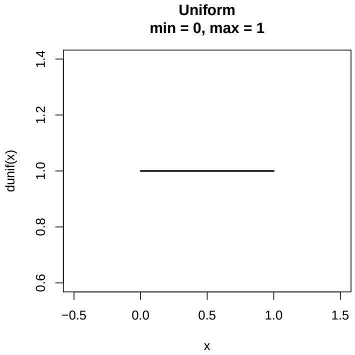


Figure 8: The Uniform distribution - known colloquially as the Anthony Davis distribution.

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:

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

## [1] 0.2058097 0.4326389 0.1622231 0.1038760 0.2073122

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

## [1]   7.944137  87.286893  41.792067 -25.983227 -90.875794
```

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`

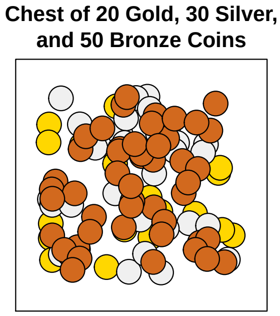
A vector of probabilities of the same length as `x` indicating how likely each outcome in "x" is. For example, to sample equally from two outcomes, you'd enter `prob = c(.5, .5)`. The first value corresponds to the first value of x and the second corresponds to the second value (etc.). The vector of 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" "Tails" "Tails" "Heads" "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.

```
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))
       )
```



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

The output of the `sample()` function above is a vector of 10 strings indicating the type of coin we drew on each sample. 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...

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

Figure 9: A typical Pinder profile.

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

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

Descriptive statistics

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.

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)`

Each of these functions takes a vector as an argument, and returns a scalar as a result. 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)
```

To calculate the mean of the data, we simply write:

```
mean(tattoos)

## [1] 24.1
```

The other descriptive statistics functions are just as easy: Let's test the `median()`, `sd()`, min(), and `max()` functions:

```
median(tattoos)

## [1] 9

sd(tattoos)
```

```
## [1] 31.32074

min(tattoos)

## [1] 2

max(tattoos)

## [1] 100
```

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).

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] 96.41325

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

## [1] 100.0143

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

## [1] -3.586747

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

## [1] 0.014292
```

Looks like the law of large numbers holds up!

Additional Tips

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, 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). We can accomplish this using indexing.

```
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
```

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 1: Determine indexing values
                      size = 3,
                      replace = F)

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

## [1]   9 100  50
```

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] 50 17 20
```

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?).

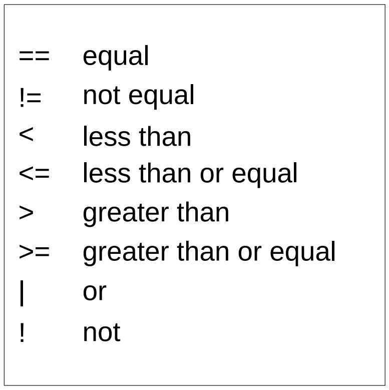
```
par(mar = rep(.1, 4))
plot(1, xlim = c(0, 1.1), ylim = c(0, 9),
     xlab = "", ylab = "", xaxt = "n", yaxt = "n",
     type = "n")


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)
```

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 10.

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.



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

Figure 10: Comparison operators in R

```
3 == 3

## [1] TRUE

3 > 5

## [1] FALSE
```

```
# 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)
```

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
```

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
```

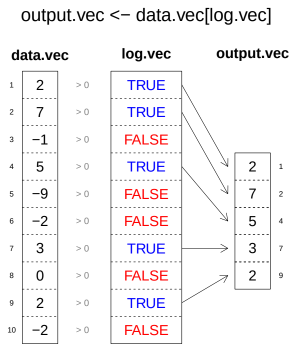
Because the fourth, sixth, and tenth values were not valid (they were negative), the final logical vector gives us FALSE values at those index values, and TRUE values for all others.

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

```
# 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
       )
```



```
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 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"
```

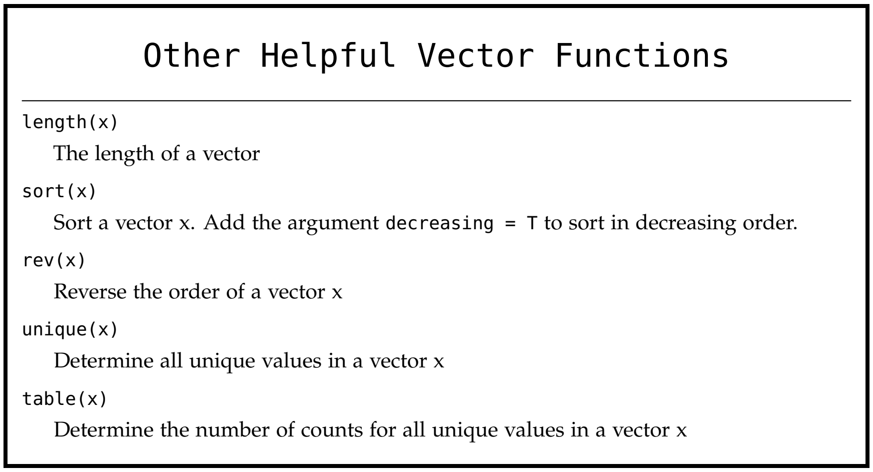
Because you can index vectors with logical vectors, I could get the same results by just indexing `pirate.names` with `improve.log`.

```
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

`unique(x)`
Determine all unique values in a vector x

`table(x)`
Determine the number of counts for all unique values in a vector x

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.

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(c(1, 1, 2, 2, 2, 4, 500))

## [1]   1   2   4 500

unique(c("a", "A", "A", "A", "b", "b", "b", "c"))

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

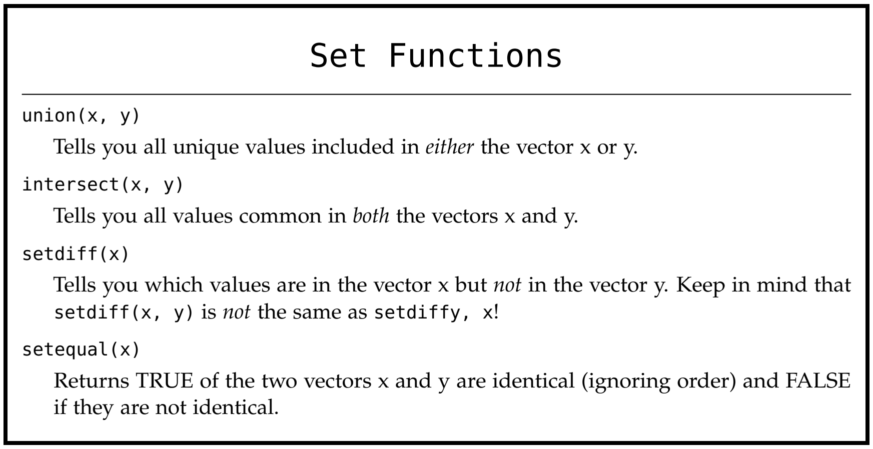
The function `table()` does the same thing as `unique()`, but goes a step further in telling you how often each of the unique values occurs:

table(x): Gives you all unique values in a vector and tells you how often each value occurs.

```
table(c(1, 1, 1, 2, 2, 5, 5, 700, 700, 1000))

##
##    1    2    5  700 1000
##    3    2    2    2    1

table(c("a", "A", "A", "A", "b", "b", "b", "c"))

##
## a A b c
## 1 3 3 1
```

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

require("RColorBrewer")

Transparent <- function(orig.col = "red", trans.val = 1, maxColorValue = 255) {

  if(length(orig.col) == 1) {orig.col <- col2rgb(orig.col)}
  if(!(length(orig.col) %in% c(1, 3))) {return(paste("length of original color mu

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

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

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)")
```

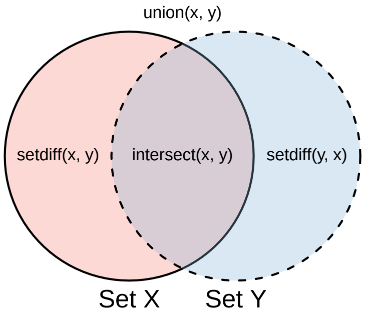


Figure 13: Common set functions in R.

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?

```
# 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
```

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

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

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 14 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"
     )
```

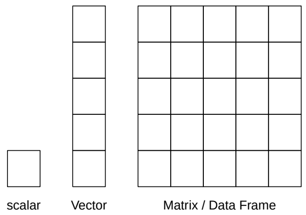


Figure 14: scalar, Vector, Matrix... ::drops mike::

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. Let's create a dataframe of fictional survey data. I'll create 5 entries for Males and 5 entries for Females. I'll then generate 10 heights from a normal distribution with mean 150 and standard deviation 10.

```
survey <- data.frame("gender" = rep(c("Female", "Male"), each = 10),
                     "height" = rnorm(20, mean = 150, sd = 10),
                     stringsAsFactors = F # don't convert strings to factors
                    )
survey # Print the dataframe

##    gender   height
## 1  Female 140.3519
## 2  Female 163.7395
## 3  Female 156.8990
## 4  Female 143.3528
## 5  Female 128.7694
## 6  Female 155.2658
## 7  Female 134.2300
## 8  Female 140.9559
## 9  Female 148.0180
## 10 Female 147.1105
## 11   Male 145.5757
## 12   Male 162.5618
## 13   Male 138.2955
## 14   Male 159.4965
## 15   Male 149.6046
## 16   Male 149.6611
## 17   Male 163.9275
## 18   Male 160.5404
## 19   Male 141.9269
## 20   Male 149.2816
```

You'll notice I included the argument `stringsAsFactors = F`, this tells R to NOT convert the strings (the Gender column) to a factor datatype. For now, don't worry about what factors are. Just know that you don't want to use them just yet!

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.

Viewing matrices and dataframes

When you start working with a new dataset loaded as a matrix or dataframe, you'll usually want to get a quick visual look at it to make sure it looks ok. There are two functions that I use to do this:

`head(x)`

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
```

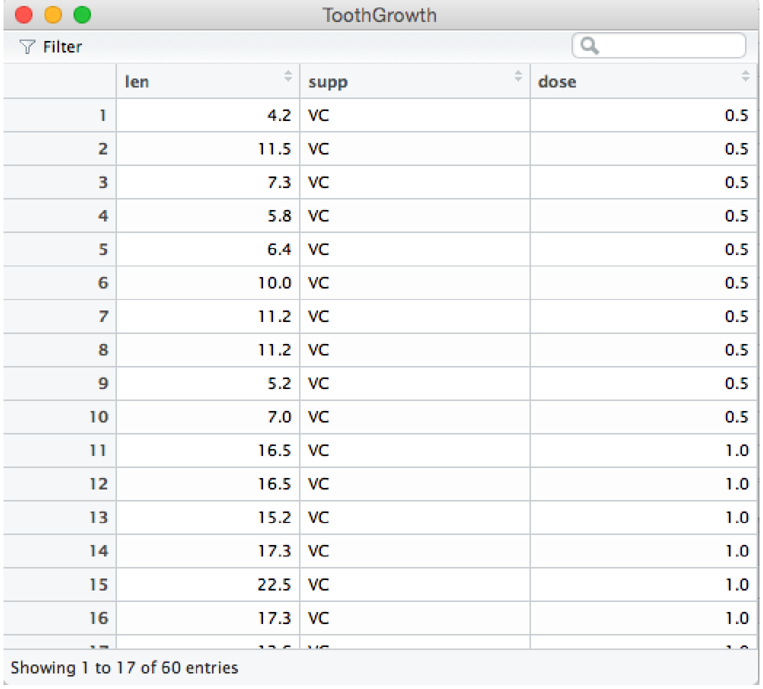
The `head()` function only shows you the first few rows of a dataframe, but usually this is enough to get a visual sense of the names of the dataframe, the number of columns, and the type of data in each column. But what if you want to see all values? You could print the entire dataframe into the console, 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(x)
```

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 15). 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.



	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 15: 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.

Loading data into R with read.table()

So far we've used either randomly generated data, or datasets preloaded 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 `Flights.txt`. This dataset contains data on all flights leaving the Houston airport in 2011. You can access this data in one of two ways: First, you can download this file from: http://nathanieldphillips.com/wp-content/uploads/2015/04/Flights.txt) and a note of its directory on your computer (on my computer, the path is `/Users/Nathaniel/Dropbox/Public/Flights.txt`). You can then load the data into R by using `read.table`:

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

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`

```
Flights <- read.table(file = "http://nathanieldphillips.com/wp-content/uploads/2015/04/Flights.txt",
                      header = T,
                      sep = "\t", # tab-delimited
                      stringsAsFactors = F
                      )
```

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(Flights)

##                  date hour minute  dep  arr dep_delay arr_delay carrier
## 1 2011-01-01 12:00:00   14      0 1400 1500         0       -10      AA
## 2 2011-01-02 12:00:00   14      1 1401 1501         1        -9      AA
## 3 2011-01-03 12:00:00   13     52 1352 1502        -8        -8      AA
## 4 2011-01-04 12:00:00   14      3 1403 1513         3         3      AA
## 5 2011-01-05 12:00:00   14      5 1405 1507         5        -3      AA
## 6 2011-01-06 12:00:00   13     59 1359 1503        -1        -7      AA
##   flight dest  plane cancelled time dist
## 1    428  DFW N576AA         0   40  224
## 2    428  DFW N557AA         0   45  224
## 3    428  DFW N541AA         0   48  224
## 4    428  DFW N403AA         0   39  224
## 5    428  DFW N492AA         0   44  224
## 6    428  DFW N262AA         0   45  224
```

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`

6: Basic Dataframe Manipulation

Chapter Goals

1. Getting basic information about dataframes: dim(), nrow(), ncol(), summary()
2. Indexing dataframes with brackets [,], and $
3. Subsettting dataframes with logical indexing and subset()
4. 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.

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:

- dim(x): Number of rows and columns in a dataframe x (returns a vector of length 2)
- nrow(x) ncol(x): How many rows or columns are there in a dataframe x (each function returns a scalar)
- summary(x): Summary of information about each column in a dataframe x

```
dim(survey) # How many rows and columns?

## [1] 20  2

nrow(survey) # How many rows?
```

```
## [1] 20

ncol(survey) # How many columns?

## [1] 2

summary(survey) # Summary information on each column

##     gender              height
##  Length:20          Min.   :128.8
##  Class :character   1st Qu.:141.7
##  Mode  :character   Median :148.6
##                     Mean   :149.0
##                     3rd Qu.:157.5
##                     Max.   :163.9
```

While you might not see the benefits of these functions now, they will become invaluable later if you conduct simulations on datasets.

Next let's start with the basics of indexing dataframes using brackets.

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

```
pirates[1:5, 1] # Give me rows 1 through 5 in column 1

## [1] 1 2 3 4 5

pirates[2:6, 2:3] # Give me rows 2 through 6 in columns 2 and 3

##      sex headband
## 2   male      yes
## 3 female      yes
## 4   male      yes
## 5   male      yes
## 6   male      yes

pirates[seq(from = 1, to = nrow(pirates), by = 20), 3] # Give me every 20th row in the 3rd column

##  [1] yes yes yes yes yes yes yes yes yes yes yes yes yes yes yes yes yes
## [18] no  yes yes yes yes no  yes yes yes yes yes yes yes yes yes yes yes
## [35] yes yes yes yes yes yes yes yes yes yes yes yes yes no  yes no
## Levels: no yes
```

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

If you want an entire row or column, you can simply leave one of the indices blank. For example, if I want the entire first row of `pirates` and all of the columns, I can simply leave the column index blank:

```
pirates[1,]

##   id    sex headband age college tattoos tchests.found parrots.lifetime
## 1  1 female      yes  35   JSSFP      18             8                9
##   favorite.pirate sword.type sword.speed
## 1      Blackbeard    cutlass  0.06389771
```

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.

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.found"    "parrots.lifetime" "favorite.pirate"
## [10] "sword.type"       "sword.speed"
```

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
```

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

```
mean(pirates$age)

## [1] 27.644

median(pirates$sword.speed)

## [1] 0.5599683
```

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
```

Let's look at the first few rows of pirates to make sure it worked:

```
head(pirates)

##   id    sex headband age college tattoos tchests.found parrots.lifetime
## 1  1 female      yes  35   JSSFP      18             8                9
## 2  2   male      yes  21    CCCC       6             5                1
## 3  3 female      yes  27    CCCC      12             8                1
## 4  4   male      yes  19    CCCC       9             8                1
## 5  5   male      yes  31    CCCC      11             2               13
## 6  6   male      yes  21    CCCC       7             1                0
##   favorite.pirate sword.type  sword.speed tatoos.per.year
## 1      Blackbeard    cutlass 0.0638977084       0.5142857
## 2      Blackbeard    cutlass 0.5601675763       0.2857143
## 3        Anicetus    cutlass 0.0005400172       0.4444444
## 4    Jack Sparrow    cutlass 3.8770396912       0.4736842
## 5    Jack Sparrow    cutlass 0.5080594239       0.3548387
## 6    Jack Sparrow    cutlass 0.6248019344       0.3333333
```

As you can see, the pirates dataframe now has a column named tattoos.per.year.

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, the following code *won't work* because R thinks that parrots.lifetime and age are totally separate objects from pirates

```
pirates$parrots.per.year <- parrots.lifetime / age # BAD CODE!
pirates$parrots.per.year <- pirates$parrots.lifetime / pirates$age # GOOD CODE!
```

However, 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, and R will assume you're referring to that object in an expression.

For example, let's repeat the pirates$parrots.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$parrots.per.year <- with(pirates, parrots.lifetime / age)
```

As you can imagine, if you're performing a long set of calculations on many columns of a dataframe, the `with()` function can save you lots of typing!

Centering and standardizing (z-score) data

Centering and standardizing are two common methods of transforming data. Centering data simply means transforming the data so that the mean is 0, while standardizing data means centering the data and dividing all data points by the standard deviation of the data. Here's how to do each:

Centering

Centering data is quite easy. All you need to do is calculate the mean of a vector, then subtract that mean from all data in the vector.

Generally, if we have a dataframe called `df`, and we want to center a column called `x`, we'd run the following code:

```
df$x.centered <- with(df, x - mean(x))
```

Let's use this method to center the age data from `pirates` - we'll call the new column `age.c`

```
pirates$age.c <- with(pirates, age - mean(age))
```

To see if this worked, let's compare the mean of `age` and `age.c`

```
mean(pirates$age)

## [1] 27.644

mean(pirates$age.c)

## [1] 1.648611e-15
```

I know what you're thinking..."But wait!!! The mean of `pirates$age.c` isn't exactly 0!!!" Don't worry, $1.6486114 \times 10^{-15}$ is so close to 0 that, for all intents and purposes, it is equal to 0 - the reason it's not *exactly* 0 is due to peculiarities about how computers represent numbers. Don't ask me why, I'm just a pirate.

Standardizing

Standardizing data is almost as easy as centering. The only difference is that, in addition to subtracting the mean from the data, we need to divide the data by its standard deviation. If you have a dataframe `df` and you want to standardize a column `x` into a new column called `x.z`, we use the following code:

```
# Create a standardized version of column x in a dataframe df
df$x.z <- with(df, (x - mean(x)) / sd(x))
```

Let's use this method to standardize the weight data from `pirates` - we'll call the new column `age.z`

```
pirates$age.z <- with(pirates, (age - mean(age)) / sd(age))
```

To see if this worked, let's compare the mean of `age`, `age.c`, and `age.z`. The mean of `age.z` should be 0 and its standard deviation should be 1:

```
c(mean(pirates$age), sd(pirates$age))

## [1] 27.644000  5.763548

c(mean(pirates$age.c), sd(pirates$age.c))

## [1] 1.648611e-15 5.763548e+00

c(mean(pirates$age.z), sd(pirates$age.z))

## [1] 2.885528e-16 1.000000e+00
```

Subsetting dataframes with logical indexing and subset()

Frequently you will want to access specific rows of a dataframe based on some criteria - this is called subsetting. For example, we may want to look just at the data from females in our survey data. To do this, we can use one of two methods: indexing with logical vectors, or the `subset()` function.

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`:

```
noparrots.log <- pirates$parrots.lifetime == 0 # Step 1: Create a logical vector
noparrots.data <- pirates[noparrots.log ,] # Step 2: Index dataframe by logical vector
head(noparrots.data) # Show the first few rows of the result

##    id    sex headband age college tattoos tchests.found parrots.lifetime
## 6   6   male      yes  21    CCCC       7             1                0
## 7   7 female      yes  31   JSSFP      11             4                0
## 11 11   male      yes  20    CCCC       7             6                0
## 12 12   male      yes  26    CCCC      13             1                0
## 15 15   male      yes  28    CCCC       9            24                0
## 16 16   male      yes  33    CCCC       8             0                0
##    favorite.pirate sword.type sword.speed tatoos.per.year parrots.per.year
## 6     Jack Sparrow    cutlass   0.6248019       0.3333333                0
## 7         Anicetus    cutlass   0.3698272       0.3548387                0
## 11      Blackbeard    cutlass   2.4274319       0.3500000                0
## 12    Jack Sparrow    cutlass   1.5420150       0.5000000                0
## 15           Hook    cutlass   0.1049165       0.3214286                0
## 16    Jack Sparrow    cutlass   0.5576000       0.2424242                0
##     age.c       age.z
## 6  -6.644 -1.15276221
## 7   3.356  0.58228025
## 11 -7.644 -1.32626645
## 12 -1.644 -0.28524098
## 15  0.356  0.06176751
## 16  5.356  0.92928874
```

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:

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

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"`

```
newdata <- pirates[pirates$age < 20 & pirates$college == "CCCC",]
head(newdata)

##    id    sex headband age college tattoos tchests.found parrots.lifetime
## 4   4   male      yes  19    CCCC       9             8                1
## 10 10   male      yes  19    CCCC      12             0                2
## 22 22 female      yes  19    CCCC      11            10                0
## 55 55 female      yes  18    CCCC       3             1                3
## 64 64   male      yes  18    CCCC       3             4                2
## 76 76   male      yes  16    CCCC      15             2                0
##    favorite.pirate sword.type sword.speed tatoos.per.year parrots.per.year
## 4     Jack Sparrow    cutlass   3.8770397       0.4736842       0.05263158
## 10      Blackbeard    cutlass   1.9394815       0.6315789       0.10526316
## 22        Anicetus    cutlass   0.1998073       0.5789474       0.00000000
## 55        Anicetus    cutlass   1.9472956       0.1666667       0.16666667
## 64    Jack Sparrow    cutlass   0.8903635       0.1666667       0.11111111
## 76    Jack Sparrow    cutlass   0.8104651       0.9375000       0.00000000
##      age.c     age.z
## 4   -8.644 -1.499771
## 10  -8.644 -1.499771
## 22  -8.644 -1.499771
## 55  -9.644 -1.673275
## 64  -9.644 -1.673275
## 76 -11.644 -2.020283
```

Looks like we have 71 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

For example, let's get the pirates data for age < 30 and college == "CCCC" and tchests.found >= 35

```
data <- subset(x = pirates,
               subset = (age < 30 & college == "CCCC" & tchests.found >= 35)
               )
head(data)

##      id    sex headband age college tattoos tchests.found parrots.lifetime
## 123 123   male      yes  21    CCCC      13            35                0
## 470 470   male      yes  24    CCCC      13            51                0
## 792 792 female      yes  28    CCCC      18            38                3
## 956 956   male      yes  25    CCCC      10            35                9
## 968 968   male      yes  26    CCCC      14            46                1
## 972 972   male      yes  16    CCCC      15            36                1
##     favorite.pirate sword.type sword.speed tatoos.per.year
## 123    Jack Sparrow    cutlass   0.1470890       0.6190476
## 470      Lewis Scot    cutlass   0.1619147       0.5416667
## 792    Jack Sparrow    cutlass   1.7545791       0.6428571
## 956    Jack Sparrow    cutlass   0.4505834       0.4000000
## 968        Anicetus    cutlass   1.2074986       0.5384615
## 972      Edward Low    cutlass   0.3370551       0.9375000
##     parrots.per.year   age.c       age.z
## 123       0.00000000  -6.644 -1.15276221
## 470       0.00000000  -3.644 -0.63224947
## 792       0.10714286   0.356  0.06176751
## 956       0.36000000  -2.644 -0.45874522
## 968       0.03846154  -1.644 -0.28524098
## 972       0.06250000 -11.644 -2.02028343
```

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:

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

```
                )
head(data)

##      id age
## 123 123  21
## 470 470  24
## 792 792  28
## 956 956  25
## 968 968  26
## 972 972  16
```

Combining indexing and descriptive statistics

Once you know how to index a dataframe to get the data vectors you want, you can then easily calculate 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:

```
# What is the mean weight of chicks on the first diet?

mean(pirates$age[pirates$college == "CCCC"]) # Using logical indexing

## [1] 24.4113

with(pirates, mean(age[college == "CCCC"])) # Logical indexing and with()

## [1] 24.4113

mean(subset(x = pirates, subset = college == "CCCC")$age) # Using subset()

## [1] 24.4113
```

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

Let's say you have a dataframe with some messed up values, how can you convert the messed up values to reasonable ones? For example, consider the following survey containing age and gender data.

```
survey <- data.frame("age" = c(49, 24, 35, 999, -10, 24),
                     "sex" = c("male", "female", "male", "male", "martian", "yes please"), stringsAsFactor
                     )


survey
```

```
##   age        sex
## 1  49       male
## 2  24     female
## 3  35       male
## 4 999       male
## 5 -10    martian
## 6  24 yes please
```

To recode numeric values, ...

```
ChickWeight[ChickWeight == 3, c(1, 2)] <- 4

## Error in ‘[<-.data.frame‘(‘*tmp*‘, ChickWeight == 3, c(1,
2), value = 4):  non-existent rows not allowed
```

A worked example: Credit default

For this example, we’ll work with a new dataset called `credit`. This dataset contains information about German loan borrowers (IVs) and whether or not the borrower defaulted on their loan (DV). To load the dataset, either download and load the data from the link `http://goo.gl/a7umut`, or simply run the following code:

```
credit <- read.table("http://nathanieldphillips.com/wp-content/uploads/2015/05/credit.csv",
                     sep = ",", header = T, stringsAsFactors = F)
```

Here is a screenshot of the dataset:

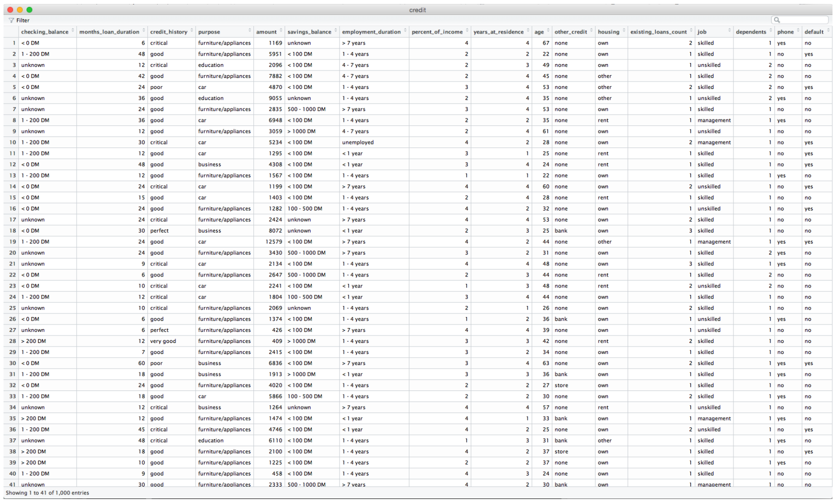
```
View(credit)
```

The dataset has 17 total columns - to see their names, execute `names(credit)`

```
names(credit)

##  [1] "checking_balance"     "months_loan_duration" "credit_history"
##  [4] "purpose"              "amount"               "savings_balance"
##  [7] "employment_duration"  "percent_of_income"    "years_at_residence"
## [10] "age"                  "other_credit"         "housing"
## [13] "existing_loans_count" "job"                  "dependents"
## [16] "phone"                "default"
```

Let’s answer 5 questions with this dataset:

1. Was there a relationship between the size of the loan and whether or not it defaulted?



	checking_balance	months_loan_duration	credit_history	purpose	amount	savings_balance	employment_duration	percent_of_income	years_at_residence	age	other_credit	housing	existing_loans_count	job	dependents	phone	default
1	< 0 DM	6	critical	furniture/appliances	1169	unknown	> 7 years	4	4	67	none	own	2	skilled	1	yes	no
2	1 - 200 DM	48	good	furniture/appliances	5951	< 100 DM	1 - 4 years	2	2	22	none	own	1	skilled	1	no	yes
3	unknown	12	critical	education	2096	< 100 DM	4 - 7 years	2	3	49	none	own	1	unskilled	2	no	no
4	< 0 DM	42	good	furniture/appliances	7882	< 100 DM	4 - 7 years	2	4	45	none	other	1	skilled	2	no	no
5	< 0 DM	24	poor	car	4870	< 100 DM	1 - 4 years	3	4	53	none	other	2	skilled	2	no	yes
6	unknown	36	good	education	9055	unknown	1 - 4 years	2	4	35	none	other	1	unskilled	2	yes	no
7	unknown	24	good	furniture/appliances	2835	500 - 1000 DM	> 7 years	3	4	53	none	own	1	skilled	1	no	no
8	1 - 200 DM	36	good	car	6948	< 100 DM	1 - 4 years	2	2	35	none	rent	1	management	1	yes	no
9	unknown	12	good	furniture/appliances	3059	> 1000 DM	4 - 7 years	2	4	61	none	own	1	unskilled	1	no	no
10	1 - 200 DM	30	critical	car	5234	< 100 DM	unemployed	4	2	28	none	own	2	management	1	no	yes
11	1 - 200 DM	12	good	car	1295	< 100 DM	< 1 year	3	1	25	none	rent	1	skilled	1	no	yes
12	< 0 DM	48	good	business	4308	< 100 DM	< 1 year	3	4	24	none	rent	1	skilled	1	no	yes
13	1 - 200 DM	12	good	furniture/appliances	1567	< 100 DM	1 - 4 years	1	1	22	none	own	1	skilled	1	yes	no
14	< 0 DM	24	critical	car	1199	< 100 DM	> 7 years	4	4	60	none	own	2	unskilled	1	no	yes
15	< 0 DM	15	good	car	1403	< 100 DM	1 - 4 years	2	4	28	none	rent	1	skilled	1	no	no
16	< 0 DM	24	good	furniture/appliances	1282	100 - 500 DM	1 - 4 years	4	2	32	none	own	1	unskilled	1	no	yes
17	unknown	24	critical	furniture/appliances	2424	unknown	> 7 years	4	4	53	none	own	2	skilled	1	no	no
18	< 0 DM	30	perfect	business	8072	unknown	< 1 year	2	3	25	bank	own	3	skilled	1	no	no
19	1 - 200 DM	24	good	car	12579	< 100 DM	> 7 years	4	2	44	none	other	1	management	1	yes	yes
20	unknown	24	good	furniture/appliances	3430	500 - 1000 DM	> 7 years	3	2	31	none	own	1	skilled	2	yes	no
21	unknown	9	critical	car	2134	< 100 DM	1 - 4 years	4	4	48	none	own	3	skilled	1	yes	no
22	< 0 DM	6	good	furniture/appliances	2647	500 - 1000 DM	1 - 4 years	2	3	44	none	rent	1	skilled	2	no	no
23	< 0 DM	10	critical	car	2241	< 100 DM	< 1 year	1	3	48	none	rent	2	unskilled	2	no	no
24	1 - 200 DM	12	critical	car	1804	100 - 500 DM	< 1 year	3	4	44	none	own	1	skilled	1	no	no
25	unknown	10	critical	furniture/appliances	2069	unknown	1 - 4 years	2	1	26	none	own	2	skilled	1	no	no
26	< 0 DM	6	good	furniture/appliances	1374	< 100 DM	1 - 4 years	1	2	36	bank	own	1	unskilled	1	yes	no
27	unknown	6	perfect	furniture/appliances	426	< 100 DM	> 7 years	4	4	39	none	own	1	unskilled	1	no	no
28	> 200 DM	12	very good	furniture/appliances	409	> 1000 DM	1 - 4 years	3	3	42	none	rent	2	skilled	1	no	no
29	1 - 200 DM	7	good	furniture/appliances	2415	< 100 DM	1 - 4 years	3	2	34	none	own	1	skilled	1	no	no
30	< 0 DM	60	poor	business	6836	< 100 DM	> 7 years	3	4	63	none	own	2	skilled	1	yes	yes
31	1 - 200 DM	18	good	business	1913	> 1000 DM	< 1 year	3	3	36	bank	own	1	skilled	1	yes	no
32	< 0 DM	24	good	furniture/appliances	4020	< 100 DM	1 - 4 years	2	2	27	store	own	1	skilled	1	no	no
33	1 - 200 DM	18	good	car	5866	100 - 500 DM	1 - 4 years	2	2	30	none	own	2	skilled	1	yes	no
34	unknown	12	critical	business	1264	unknown	> 7 years	4	4	57	none	rent	1	unskilled	1	no	no
35	> 200 DM	12	good	furniture/appliances	1474	< 100 DM	< 1 year	4	1	33	bank	own	1	management	1	yes	no
36	1 - 200 DM	45	critical	furniture/appliances	4746	< 100 DM	< 1 year	4	2	25	none	own	2	unskilled	1	no	yes
37	unknown	48	critical	education	6110	< 100 DM	1 - 4 years	1	3	31	bank	other	1	skilled	1	yes	no
38	> 200 DM	18	good	furniture/appliances	2100	< 100 DM	1 - 4 years	4	2	37	store	own	1	skilled	1	no	yes
39	> 200 DM	10	good	furniture/appliances	1225	< 100 DM	1 - 4 years	2	2	37	none	own	1	skilled	1	yes	no
40	1 - 200 DM	9	good	furniture/appliances	458	< 100 DM	1 - 4 years	4	3	24	none	own	1	skilled	1	no	no
41	unknown	30	good	furniture/appliances	2333	500 - 1000 DM	> 7 years	4	2	30	bank	own	1	management	1	no	no

Showing 1 to 41 of 1,000 entries

Figure 16: Screenshot of the credit dataset.

```
amount.default <- credit$amount[credit$default == "yes"]
amount.nodefault <- credit$amount[credit$default == "no"]

summary(amount.default)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     433    1352    2574    3938    5142   18420

summary(amount.nodefault)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     250    1376    2244    2985    3635   15860
```

We calculated the median loan size separately for people whose loans defaulted and those whose loans did not default. The loan amounts of loans that defaulted (median of 2574) tended to be a bit larger than those that did not (median of 2244). However, looking at the full amount distributions (see Figure 17), it is unclear if the difference is really very meaningful.

```
amount.default <- credit$amount[credit$default == "yes"]
amount.nodefault <- credit$amount[credit$default == "no"]

require(beanplot)

## Loading required package:  beanplot

beanplot(amount ~ default, data = credit,
         col = c("white", gray(.8), gray(.8), "black"),
         names = c("No", "Yes"),
         main = "Loan size by default",
         xlab = "Did the loan default?",
         ylab = "Loan size (log-transformed)",
         what = c(1, 1, 1, 0)
         )

## log="y" selected
```

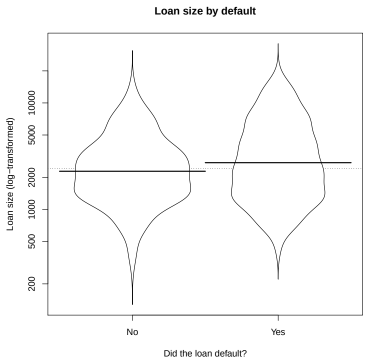
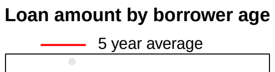


Figure 17: Distributions of loan sizes separated for loans that defaulted and those that did not.

2. Was the age of the borrower related to the loan amount?

```
summary(credit$amount[credit$age <= median(credit$age)])

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     276    1364    2310    3193    3924   18420

summary(credit$amount[credit$age > median(credit$age)])

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     250    1370    2324    3354    4042   15940
```

```
# Main Plot
plot(credit$age, credit$amount,
     pch = 16, col = gray(.5, alpha = .2),
     main = "Loan amount by borrower age",
     xlab = "Borrower Age", ylab = "Loan Amount (DM)"
     )

par(xpd=NA)
segments(25, 20000, 35, 20000, col = "red", lwd = 2)
text(50, 20000, "5 year average")

# Create factor from age
age.cut <- cut(credit$age, breaks = seq(20, 70, 5))

# Determine mean loan by age factor
amount.cut <- tapply(credit$amount, age.cut, mean)

# Add mean lines
lines(seq(22.5, 67.5, 5), amount.cut,
      col = "red", lwd = 2, type = "b", pch = 16)
```



```
cor(credit$age, credit$amount)

## [1] 0.03271642
```

To answer this, we separately calculated the median loan amount for borrowers below and above the median age (of 33). Borrowers below the median age had a median loan amount of 2310, while borrowers above the median age had a median loan amount of 2324. This suggests that age was unrelated to loan amount. Examining the scatterplot in Figure 18, we do not find strong evidence for an effect of borrower age on loan amount.

3. Was there a relationship between whether or not someone had a phone and whether or not their loan defaulted?

```
with(credit, table(phone, default))

##      default
## phone  no yes
##   no  409 187
##   yes 291 113

with(credit[credit$phone == "yes",], mean(default == "yes"))

## [1] 0.279703

with(credit[credit$phone == "no",], mean(default == "yes"))

## [1] 0.3137584
```

We separately the proportion of people who defaulted on their loans separately between those who own a phone and those who do not. We found that people without a phone were slightly more likely to default (31.38%) than people with a phone (27.97%) (a `mosaic.plot` of the data is presented in margin Figure).

```
require(RColorBrewer)
with(credit, mosaicplot(table(phone, default),
                  main = "Phone Ownership and Loan Default",
                  xlab = "Own Phone?",
                  ylab = "Loan Default?", color = brewer.pal(12, "Set3")[5:4]
     ))

defper.withphone <- mean(credit$default[credit$phone == "yes"] == "yes")
defper.nophone <- mean(credit$default[credit$phone == "no"] == "yes")

text(mean(credit$phone == "no") / 2,
     defper.nophone / 2,
     paste(100 * round(defper.nophone, 2), "%", sep = "")
     )

text(1 - mean(credit$phone == "yes") / 2,
     defper.withphone / 2,
     paste(100 * round(defper.withphone, 2), "%", sep = "")
     )
```

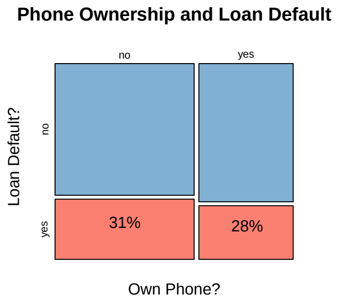


Figure 19: Mosaic plot of the contingency between phone ownership and loan defaults in the `credit` dataset. People who own a phone are slightly less likely to default on their loans that people who do not own a phone.

Additional Tips

- If you want to change the names of columns in a dataframe, you can do this by reassigning elements of the `names()` function. For example, let's change the names of the first two columns of our dataframe `survey` to "Sex" and "Height.cm"

```
names(survey)

## [1] "age" "sex"

names(survey)[1:2] <- c("Sex", "Height.cm")
names(survey)
```

```
## [1] "Sex"       "Height.cm"
```

7: Plotting Basics

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.

Figure 20: 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.

Plotting in R works like putting paint on a canvas. You start by creating a canvas and plotting the main elements using a *high-level* plotting command. In these high-level plotting commands, you specify things like the x and y coordinates of the plot, the plot titles, and the main data in the plot. Next, you use *low-level* plotting commands to sequentially add as many additional individual elements as you'd like, from lines to arrows to text. Once you are done, you can export the plot as a jpg or pdf file. In the next section, we'll cover the most common high-level plotting functions

High-level plotting functions

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()` command creates a scatterplot from two vectors x and y:

`plot(x, y)`: Create a scatterplot from two vectors x and y.
  `main`: Title of plot
  `xlab, ylab`: axes labels
  `xlim, ylim`: Limits of axes
  `xaxt, yaxt`: Set to `"n"` to remove the axes
  `cex`: Size of the plotting points
  `pch`: Type of plotting points (see `?points`)

`plot()`

`x, y`

Two vectors of data on the x and y-axes

`main`

The title of the plot

`xlab, ylab`

Labels for the x and y-axes.

`xlim, ylim`

A 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 ylimitx to [50, 60].

`col`

The color of the plotting points. For example `col = "red"` will create red plotting points.

`pch`

An integer indicating the type of plotting symbols (see `?points` and section below), or a string specifying symbols as text.

`cex`

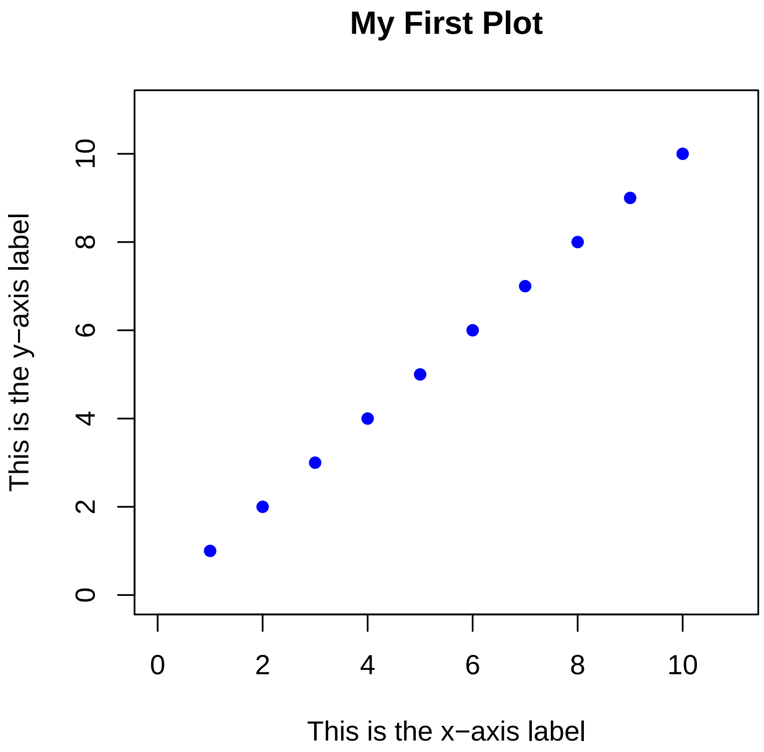
The size of the symbols (from 0 to Inf). The default size is 1.

`type`

The type of plot. Use `"p"` for points (the default), `"l"` for lines, `"b"` for points and lines, and `"\n"` for no plotting

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].

```r
plot(x = 1:10,
     y = 1:10,
     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 (Filled circle)
     cex = 1, # Size of the symbols,
     type = "p" # Plot
     )
```



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.

Symbol types: pch

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 21 in the margin.

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

```
par(mar  = rep(0, 4))

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

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

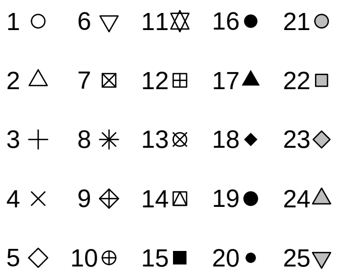


Figure 21: The symbol types associated

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.

Other high-Level plotting commands

While `plot()` is the most widely used high-level plotting command, there are several (perhaps even hundreds) of additional ones. I'll briefly highlight a few additional ones that you may wish to use

Histograms `hist()`

The function `hist()` is a high-level plotting command that creates (wait for it...) a histogram. Here are the main arguments for `hist()`:

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

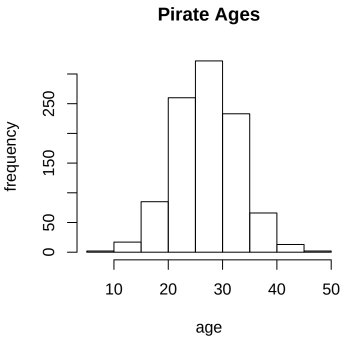


Figure 22: Histogram of age data from `pirates`

`hist()`

`x`

A vector of 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`

The color of the filling of the bars. (e.g.; `col = "red"`)

`border`

The color of the border of the bars. (e.g.; `border = "green"`)

`probability`

A logical value indicating whether to plot the results as probabilities (the default is FALSE)

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

Other standard plotting arguments

You can see an example of a histogram in the margin Figure 22.

Boxplots `boxplot()`

Boxplots aren't used so often anymore (for reasons that I'll show you shortly), but I think it's good to know how to make them, even if it's just for historical purposes. 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")`

`horizontal`

A logical value indicating whether to plot the boxes horizontally.

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

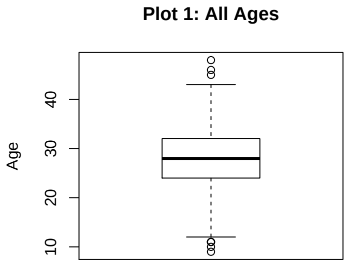


Figure 23: Plotting data from a single vector

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

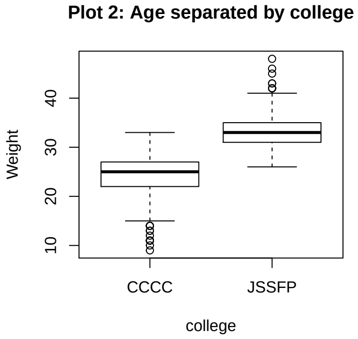


Figure 24: Plotting data as a function of levels of an independent variable using the `y ~ x` formula notation.

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.

Beanplots: `beanplot()`

The last high-level plotting I want to show you is the `beanplot()` function. This function creates a beanplot, which (like boxplots and histograms), shows you a distribution of sample data. However, as you can see in Figure **??**, they look much, much cooler than a boxplot or histogram. What’s really great about beanplots is that they show you a combination of three elements: raw data, smoothed distribution lines, and group averages. This means that you can quickly detect outliers, multiple modes, or missing data much better than you can with boxplots.

To use the `beanplot()` function, you first need to download the `beanplot` package:

```
install.packages("beanplot")
```

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

`beanplot()`

`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, data = survey`

`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.

`what`

A vector of four Boolean (0 or 1) values indicating what to plot in the following order: the total average line, the beans, the bean average, and the beanlines. For example, to plot everything, use `what = c(1, 1, 1, 1)`. To plot just the beans, use `what = c(0, 0, 1, 0)`

`color`

The colors in the plot. A vector of up to four colors can be used representing the areas of the beans, the lines inside the beans, the lines outside the beans, and the average line per bean. If you want to make each bean a different color, you have to specify a list of separate color vectors, one for each bean. Look at my code in Figure **??** for a way to do this using `lapply`.

`names`

A vector of names of the beans.

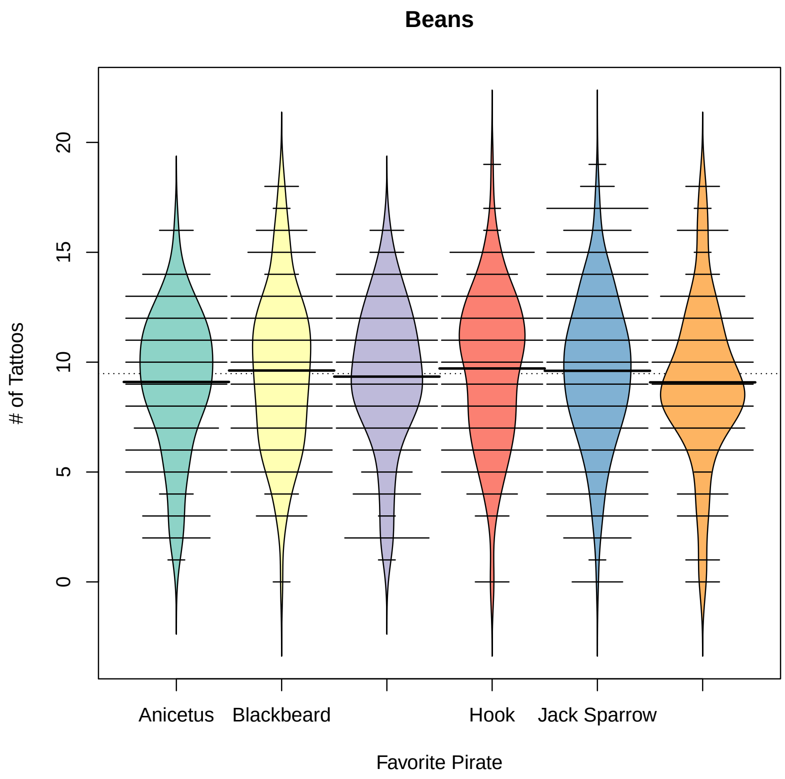
`overallline`

A method for determining the overall line (either `"mean"`or `"median"`)

The code for creating beanplots is very similar to the code for boxplots. Let's create a beanplot showing the distribution of how many tattoos pirates have as a function of their favorite pirate:

```
require("beanplot")
require("RColorBrewer")
```

```
bean.cols <- lapply(brewer.pal(6, "Set3"),
                    function(x) {return(c(x, "black", "black", "black"))})

beanplot(tattoos ~ favorite.pirate,
         data = pirates,
         main = "Beans",
         xlab = "Favorite Pirate",
         ylab = "# of Tattoos",
         col = bean.cols ,
         lwd = 1,
         what = c(1, 1, 1, 1), log = ""
         )
```



One major new argument is what, which dictates what exactly is plotted. You specify what using a vector of four Boolean (0 or 1) values. In the plot in Figure **??**, I've set all values to 1 which means that the function will include all four plot elements. If you'd like to remove certain elements, like the individual lines or the average lines, you can remove them by replacing the respective 1s to 0s.

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.

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

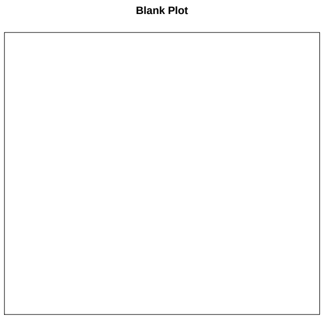


Figure 25: A blank plot. Useful to start with before adding elements with low-level plotting commands. Just make sure to set the axis limits to values that make sense for your future data.

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 25 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

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

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

# Create a blank plot
plot(x = 1, y = 1, xlab = "Age", ylab = "Number of Tattoos",
     type = "n", main = "Even More Chicken Weights",
     xlim = c(1, 50), ylim = c(0, 20))

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

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

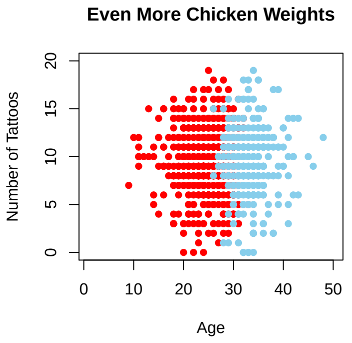


Figure 26: Adding additional points to an existing plot with points()

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 26, I use the points function to plot data from pirates, where pirates from Captain Chunk's Canon Crew are plotted 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:

```
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)
```

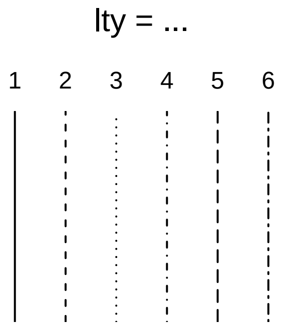


Figure 27: Line types generated from arguments to lty.

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
```

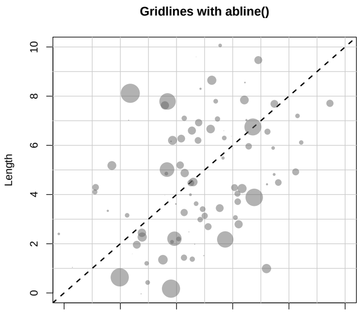
You can easily use abline() to add gridlines to plots by entering vectors in the `h` and `v` arguments. For example, to add gridlines to a plot at x-values and y-values from 0 to 10 in steps of 1, you'd enter

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

```
# Create a blank plot
plot(x = 1, y = 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))
       )
```

In margin Figure 28 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 (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

```r
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 29 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()`:

```
# 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
     )
```

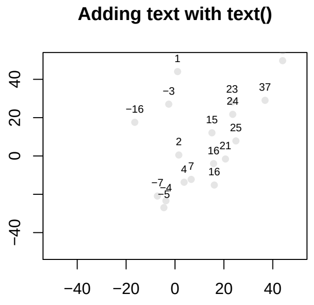


Figure 29: Adding text to a plot with text().

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
     )
```

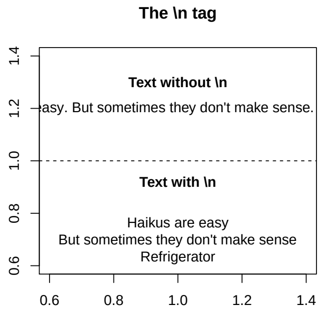


Figure 30: Using the `"\n"` tag to plot text on separate lines.

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.

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

## [1] 21.00166

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

## [1] "The mean of the group is 21.0016609949259"

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

## [1] "The mean of the group is 21"
```

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`, ...

```
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"
       )
```

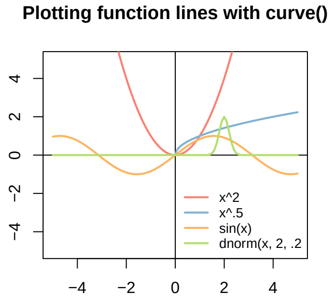


Figure 31: Using `curve()` to easily create lines of functions

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

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

```
# 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"
       )
```

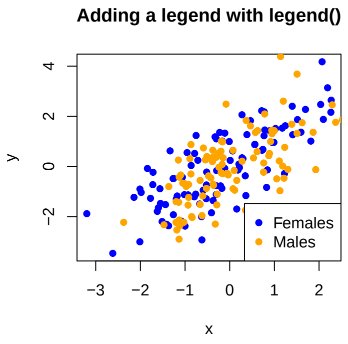


Figure 32: Creating a legend labeling the symbol types from different groups

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:

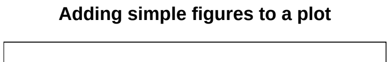
```
legend("bottomright", # Put legend in bottom right of graph
       legend = c("Females", "Males"), # Names of groups
       col = c("blue", "orange"), # Colors of symbols
       pch = c(16, 16) # Point types
       )
```

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

```
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
         )
```

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.

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
```

```r
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.

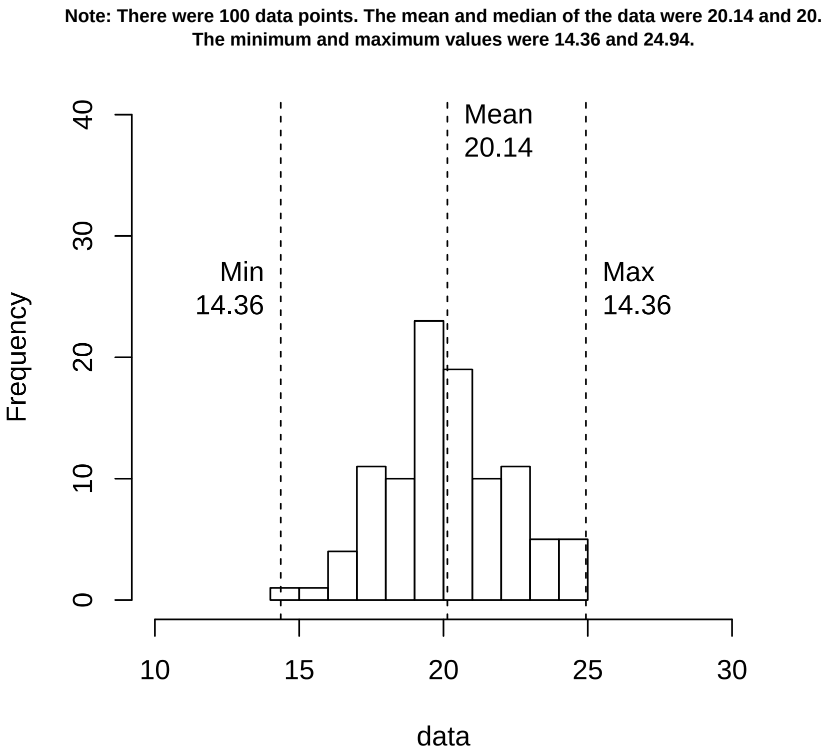
A worked example: Creating a plot with automated numeric labels

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.

```r
# 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.

8: Customizing Plots

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] "gray93"          "lightyellow"     "grey74"
##  [4] "mediumslateblue" "darkseagreen"    "grey73"
##  [7] "ivory"           "ivory3"          "sandybrown"
## [10] "dodgerblue2"
```

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 33.

```r
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"
     )
```

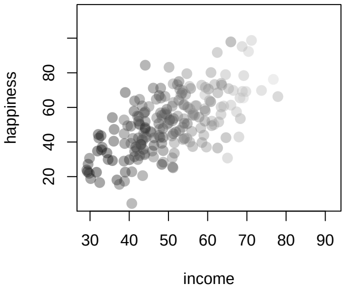


Figure 33: Using the `gray()` function to easily create shades of gray in plotting symbols based on numerical data.

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:

```r
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"
```

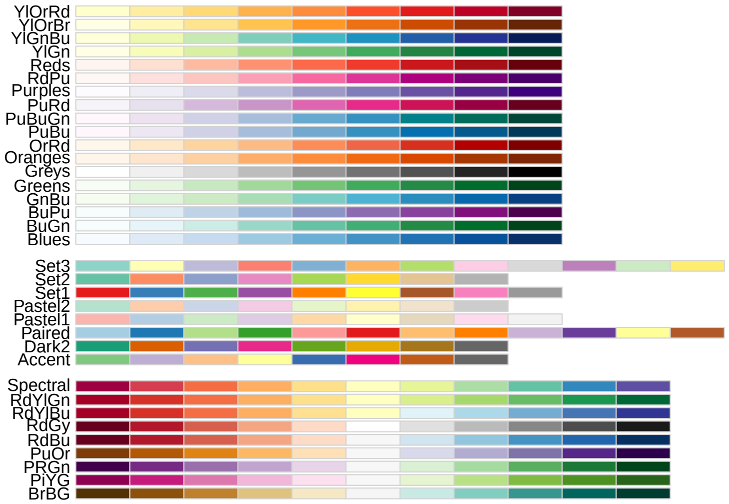
Now, if you're not used to interpreting colors as RGB values, you may gloss over this section and think you'll never use it. However, look at this chapter's *Additional Tips* for a really cool method of 'stealing' the exact color from *anything* on your computer screen and using `rgb()` to then use that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

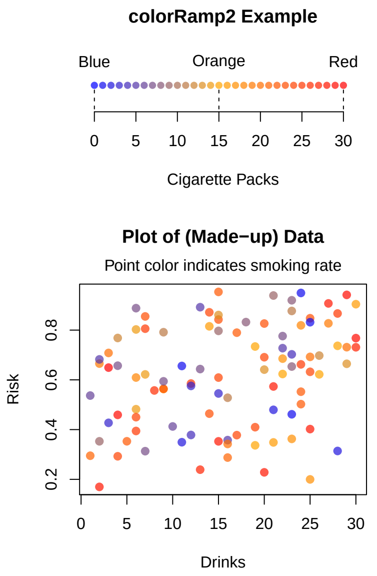
`colorRamp2` allows you to do exactly this. The function has three arguments:

- `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:

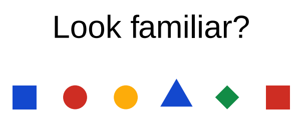
```
smoking.colors <- colorRamp2(breaks = c(0, 15, 30),
                             colors = c("blue", "orange", "red"),
                             transparency = .3
                             )

smoking.colors(0) # Equivalent to blue

## [1] "#0000FFB3"

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

## [1] "#FF6E00B3"
```

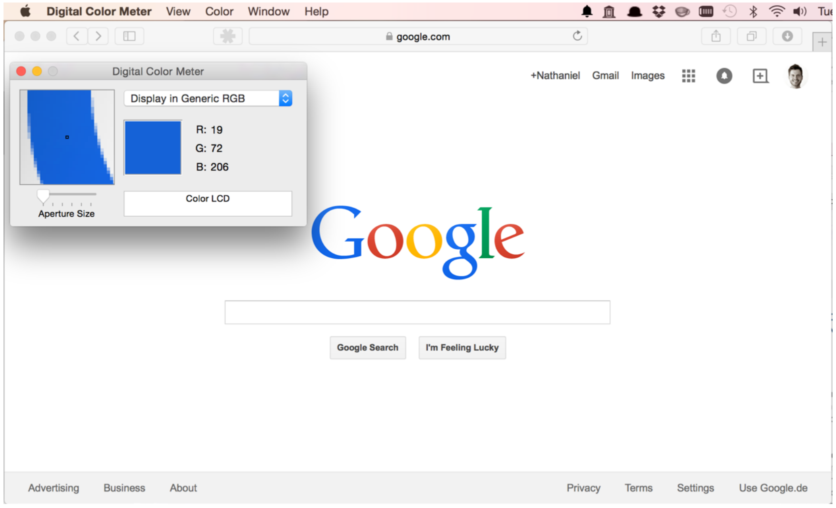
To see this function in action, check out the the margin Figure for an example, and check out the help menu `?colorRamp2` for more information and examples.

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)
```





Display in Generic RGB

R: 19
G: 72
B: 206

Color LCD

Plot margins

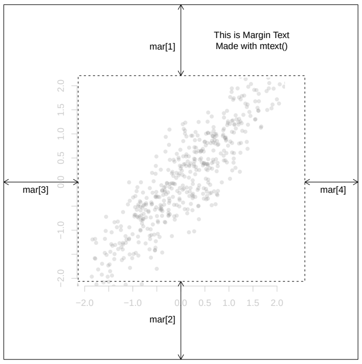
All plots in R have margins surrounding them that separate the main plotting space from the area where the axes, labels and additional text lie.. To visualize how R creates plot margins, look at margin Figure .

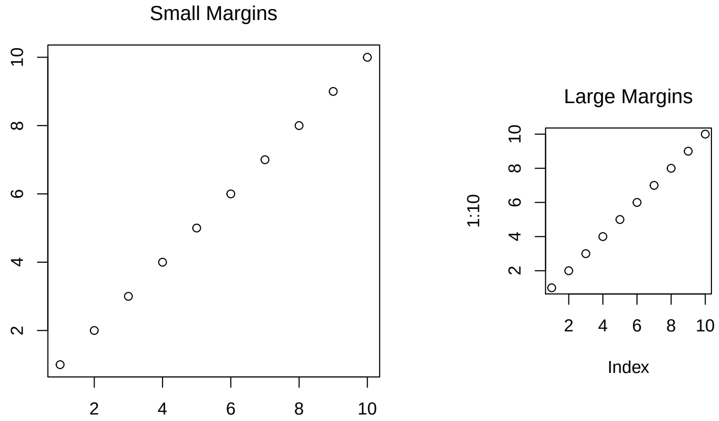
You can adjust the size of the margins by specifying a margin parameter using the syntax `par(mar = c(a, b, c, d))` before you execute your first high-level plotting function, where a, b, c and d are the size of the margins on the bottom, left, top, and right of the plot. Let's see how this works by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





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.

The default value for `mar` is `c(5.1, 4.1, 4.1, 2.1)`

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:

```
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)

}
```

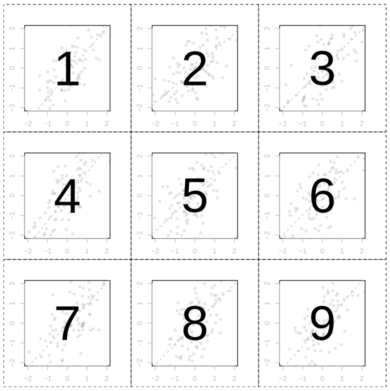


Figure 35: A matrix of plotting regions created by `par(mfrow = c(3, 3))`

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)
```

- `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 36)

```
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)
```

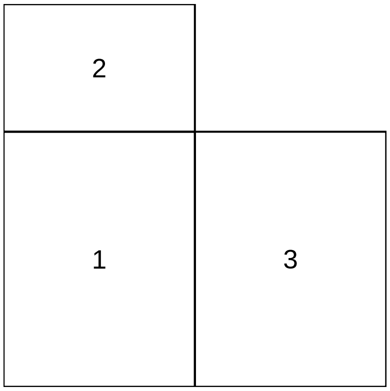
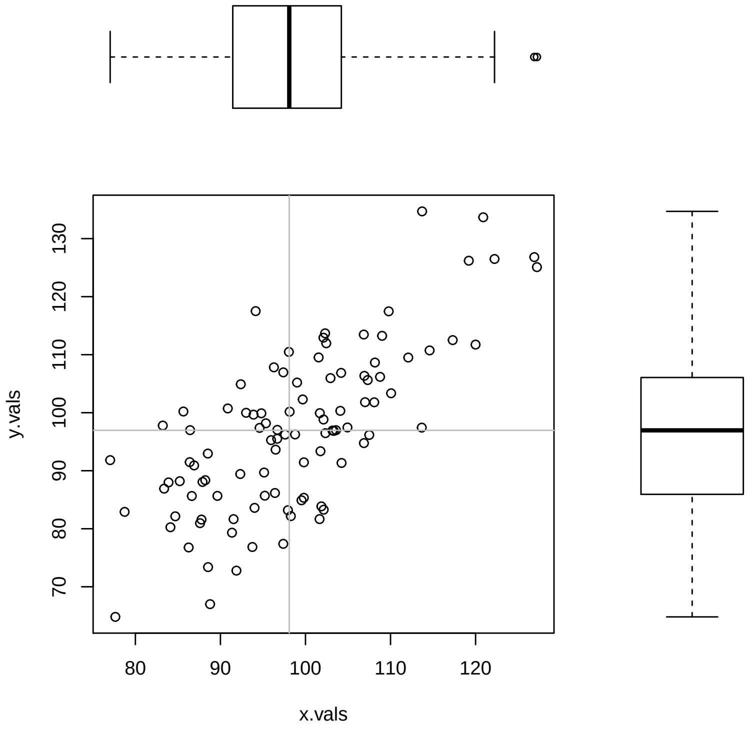


Figure 36: A plotting layout created by setting a layout matrix and specific heights and widths.

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)
```

```
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
```

```
##  [1] ".Keyboard"                       "Andale Mono"
##  [3] "Apple Braille"                   "AppleMyungjo"
##  [5] "Arial Black"                     "Arial"
##  [7] "Arial Narrow"                    "Arial Rounded MT Bold"
##  [9] "Arial Unicode MS"                "Brush Script MT"
## [11] "Comic Sans MS"                   "Courier New"
## [13] "DIN Alternate"                   "DIN Condensed"
## [15] "Georgia"                         "Impact"
## [17] "Khmer Sangam MN"                 "Lao Sangam MN"
## [19] "Microsoft Sans Serif"            "Myanmar Sangam MN"
## [21] "Tahoma"                          "Times New Roman"
## [23] "Trebuchet MS"                    "Verdana"
## [25] "Webdings"                        "Wingdings"
## [27] "Wingdings 2"                     "Wingdings 3"
## [29] "Helvetica LT Std  steevo harvie" "Helvetica World"
## [31] "HelvFB"                          "HelvFE"
## [33] "Helvetica-Black-SemiBold"        "Helvetica"
## [35] "Helvetica-Condensed-Black-Se"    "Helvetica-Condensed-Light-Li"
## [37] "Helvetica-Condensed-Light-Light" "Helvetica-Condensed-Thin"
## [39] "Helvetica-Conth"                 "Helvetica-Light-Light-Italic"
## [41] "Helvetica-Normal"                "HelveticaExt-No"
## [43] "HelveticaExtO 2"                 "HelveticaExtO 3"
## [45] "HelveticaExtO 4"                 "HelveticaInserat-Roman-SemiB"
## [47] "HelveticaInserat-Roman-SemiBold" "HelveticaNeueLT Com 55 Roman"
## [49] "HelveticaNeueLT Com 57 Cn"       "HelveticaNeueLT Com 53 Ex"
```

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
    )

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

dev.off()

## pdf
##   2

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

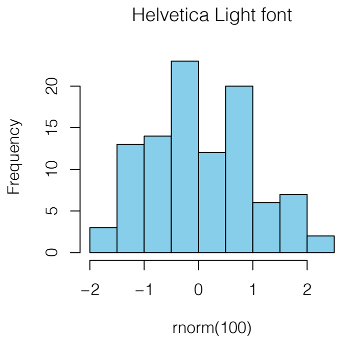


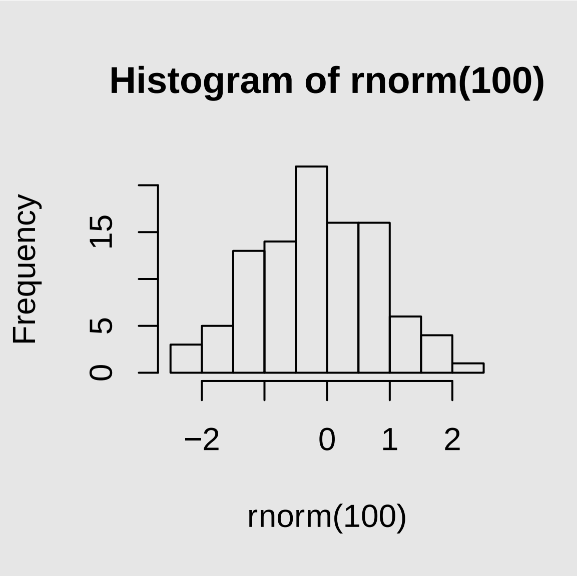
Figure 37: Plot created with Helvetica Light font (see the main text for plotting code).

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

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 38 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")

parrot.data <- data.frame(
  "parrots" = 0:6,
  "female" = c(200, 150, 100, 175, 55, 25, 10),
  "male" = c(150, 125, 180, 242, 10, 62, 5)
  )

n.data <- nrow(parrot.data)

par(bg = rgb(61, 55, 72, maxColorValue = 255),
    mar = c(8, 6, 6, 3)
    )

plot(1, xlab = "", ylab = "", xaxt = "n",
     yaxt = "n", main = "", bty = "n", type = "n",
     ylim = c(0, 250), xlim = c(.5, n.data + .5)
     )

abline(h = seq(0, 250, 50), lty = 3, col = gray(.95), lwd = 1)

mtext(text = seq(50, 250, 50),
      side = 2, at = seq(50, 250, 50),
      las = 1, line = 1, col = gray(.95))

mtext(text = paste(0:(n.data - 1), " Parrots"),
      side = 1, at = 1:n.data, las = 1,
      line = 1, col = gray(.95))


female.col <- gray(1, alpha = .7)
male.col <- rgb (226, 89, 92, maxColorValue = 255, alpha = 220)

rect.width <- .35
rect.space <- .04

rect(1:n.data - rect.width - rect.space / 2,
     rep(0, n.data),
     1:n.data - rect.space / 2,
     parrot.data$female,
     col = female.col, border = NA
     )

rect(1:n.data + rect.space / 2,
     rep(0, n.data),
     1:n.data + rect.width + rect.space / 2,
     parrot.data$male,
     col = male.col, border = NA
     )

legend(n.data - 1, 250, c("Male Pirates", "Female Pirates"),
       col = c(female.col, male.col), pch = rep(15, 2),
       bty = "n", pt.cex = 1.5, text.col = "white"
       )

mtext("Number of parrots owned by pirates", side = 3,
      at = n.data + .5, adj = 1, cex = 1.2, col = "white")

mtext("Source: Drunken survey on 22 May 2015", side = 1,
      at = 0, adj = 0, line = 3, font = 3, col = "white")

dev.off()


## pdf
##   2


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

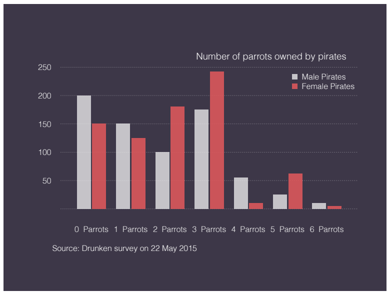


Figure 38: Use `par(bg = my.color)` before creating a plot to add a colored background. The design of this plot was inspired by `http://www.vox.com/2015/5/20/8625785/expensive-win`

9: *Advanced dataframe manipulation*

Chapter Goals

1. Grouped aggregation with aggregate() and dplyr
2. Merging datasets with merge()
3. Common data management tasks: rescaling, recoding

In this chapter, we'll cover how to do advanced dataframe manipulation. You'll learn how to recode values of a dataframe and quickly and easily calculate summary data from a dataframe,

Recoding values in a vector

Let's say that you conducted an online survey where you asked 100 people some basic demographic information and then asked them how happy they are with their life on a scale from 1 to 8. When you look at the dataset, you realize that many people gave invalid answers to the questions. For example, some people said they had a negative income and had a negative number of siblings. How can you fix these data? In this chapter, we'll cover the basics of data-recoding

There are three basic steps to recoding values in a vector:

3 Steps to recoding values

1. Copy the original column (`original`) to a new column with a new name (`new`) using assignment.
2. Create a logical vector indicating which values of `new` should be converted.
3. Index `new` by the logical vector and assign the updated value to it.

3 Steps to recoding values

1. Copy the original column (`original`) to a new column with a new name (`new`) using assignment.
2. Create a logical vector indicating which values of `new` should be converted.
3. Index `new` by the logical vector and assign the updated value to it.

Let's start with a dummy example where we take the integers from 1 to 10, then convert all values greater than 5 to 999

```
original <- 1:10

new.vec <- original # Step 1
log.vec <- new.vec > 5 # Step 2
new.vec[log.vec] <- 999 # Step 3

new.vec

##  [1]   1   2   3   4   5 999 999 999 999 999
```

As you can see, our new vector new.vec is identical to the integers 1 to 10 up until the value of 6, where all values have been converted to 999.

Recoding values from the pirate survey

Let's start by using the dataset pirate_survey_werrors.txt which is stored at http://nathanieldphillips.com/wp-content/uploads/2015/05/pirate_survey_werrors.txt. This is a tab-delimited file containing results from 1,000 pirates.

To load the data, execute the following commands:

```
pirates.errors <- read.table("http://nathanieldphillips.com/wp-content/uploads/2015/05/pirate_survey_withe
```

Unfortunately, some pirates decided to supply some invalid answers. We'll start with the sex variable. To see the values that are currently in this column, we'll run table():

```
table(pirates.errors$sex)

##
## depends on who is offering                     female
##                          1                        466
##                       male                      other
##                        490                         41
##        sure I'll have some                yes please!
##                          1                          1
```

Very funny guys. There should only be three valid responses to this question: female, male, and other/NA. Let's recode all the other values to NA. We'll do this in three steps:

1. Copy the values of sex into a new vector called sex.r standing for "sex recoded"

2. Create a logical vector called log.vec that indicates which values of sex.r are valid - that is, in the set of values c("female", "male", "other/NA")

3. Assign a value of NA to all values of sex.r for which log.vec is FALSE.

```
# Step 1: Copy sex to a new column called sex.r
pirates.errors$sex.r <- pirates.errors$sex

# Step 2: Create a logical vector indicating which values are valid
log.vec <- pirates.errors$sex.r %in% c("female", "male", "other/NA")

# Step 3: Recode all invalid values to NA
pirates.errors$sex.r[log.vec == FALSE] <- NA # Step 3
```

Let's make sure this worked by looking at the values of sex.r. Hopefully, we will only see valid entries now:

```
table(pirates.errors$sex.r, useNA = "ifany")

##
## female   male   <NA>
##    466    490     44
```

As you can see, our new column sex.r only contains valid entries, so it looks like our recoding did what we wanted.

Let's do the same with age and tattoos. The column age should be an integer between 18 and 99, and the values of tattoos should be an integer between 0 and a maximum of (let's say) 100. Let's start by looking at the current values of each column using the table() function:

Note that the table() function does not (by default), show NA values. To see how many NA values are in the dataset, you can include the argument useNA = "ifany" as in table(pirates.errors$sex.r, useNA = "ifany")

```
table(pirates.errors$age)

##
##   -99     0     9    10    11    12    13    14    15    16    17    18
##     4     1     1     1     5     1     2     4     5     7     5    19
##    19    20    21    22    23    24    25    26    27    28    29    30
##    20    33    35    38    55    73    52    63    55    58    69    70
##    31    32    33    34    35    36    37    38    39    40    41    42
##    57    56    40    45    32    32    12    10     5     5     6     4
##    43    45    46    48   500   999 12345
##     2     1     1     1     2     4     9

table(pirates.errors$tattoos)

##
##   -10     0     1     2     3     4     5     6     7     8     9    10
##     2     8     5    13    24    24    39    60    73   124   119   110
##    11    12    13    14    15    16    17    18    19 1e+06
##   128    91    73    46    21    15     9     6     2     4
```

We can see some problems here: age has some negative values and a few very large values. Further, tattoos has a few negative values and a few values greater than 100. Let's convert all these troublesome values of age to NA.

```
# Step 1: Copy age to a new column age.r
pirates.errors$age.r <- pirates.errors$age

# Step 2: See which values of age.r are valid
age.valid <- pirates.errors$age.r >= 18 & pirates.errors$age.r <= 99 # Step 2

# Step 3: Recode non-valid values to NA
pirates.errors$age.r[age.valid == FALSE] <- NA # Step 3
```

Now let's do the same for tattoos:

```
# Step 1: Copy tattoos to a new column tattoos.r
pirates.errors$tattoos.r <- pirates.errors$tattoos

# Step 2: See which values of tattoos.r are valid
tattoos.valid <- pirates.errors$tattoos.r %in% 0:100 # Step 2

# Step 3: Recode non-valid values to NA
pirates.errors$tattoos.r[tattoos.valid == FALSE] <- NA # Step 3
```

Now, let's look at the frequency tables of these recoded values:

```
table(pirates.errors$age.r, useNA = "always")

##
##   18   19   20   21   22   23   24   25   26   27   28   29   30   31   32
##   19   20   33   35   38   55   73   52   63   55   58   69   70   57   56
##   33   34   35   36   37   38   39   40   41   42   43   45   46   48 <NA>
##   40   45   32   32   12   10    5    5    6    4    2    1    1    1   51

table(pirates.errors$tattoos.r, useNA = "always")

##
##    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14
##    8    5   13   24   24   39   60   73  124  119  110  128   91   73   46
##   15   16   17   18   19 <NA>
##   21   15    9    6    2   10
```

In these examples, we converted invalid values to NA. Of course, there's no reason why we couldn't recode the values to other values. For example, we might recode values of `tattoos` greater than 50 to the maximum possible value of 50. To do this, we can just change the assignment in Step 3 to a value of `50` instead of NA.

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.binned`, 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, 100, 10)` as the `breaks` argument:

```
pirates$age.cut <- cut(x = pirates$age,  # The raw data
                       breaks = seq(10, 100, 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.cut")])

##   age age.cut
## 1  35 (30,40]
## 2  21 (20,30]
## 3  27 (20,30]
## 4  19 (10,20]
## 5  31 (30,40]
## 6  21 (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.cut)

##
##  (10,20]  (20,30]  (30,40]  (40,50]  (50,60]  (60,70]  (70,80]  (80,90]
##      102      582      299       15        0        0        0        0
## (90,100]
##        0
```

Grouped aggregation

As you can see, we have quite a bit of data in our pirates dataset. Many of the questions we might want to answer with this dataset have to do with comparisons between groups. For example, "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 of the data

`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. 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 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 24.4113
## 2   JSSFP 33.3168
```

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 39)

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

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

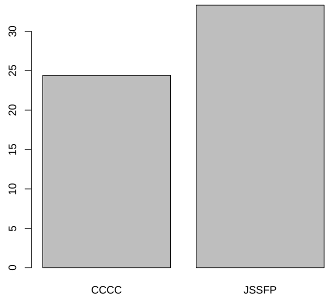


Figure 39: Barplot showing the mean age for each pirate college

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, let's use `aggregate()` to now get the median age for all combinations of college and headband:

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

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 new package that allows you to do data 'wrangling' (manipulating datasets) quickly and easily. In this section, we'll go over a very brief overview of how the functions in dplyr work. 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.

dplyr works by combining objects (dataframes and columns in dataframes), functions (mean, median, etc.), and *verbs* (special commands in `dplyr`). In between these commands is a new operator called the *pipe* which looks like this: `%>%`. The pipe simply tells R that you want to continue executing some functions or verbs on the object you are working on. You can think about this pipe as meaning 'and then...'

To aggregate data with dplyr, your code will look something like the following code. In this example, assume that the 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, where each row is a college, and each columns is a different summary statistic of some data across all flights. Specifically, let's create 5 columns: age.med, The median age of pirates from that college, age.min: The minimum age of pirates from that college, sword.speed.med: The median sword speed of pirates from that college, and tchests.found.mean: The average number of treasure chests found of pirates from that college

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:

```
require(dplyr)
college.agg <- pirates %>% # Define dataframe, THEN...
                group_by(college) %>% # 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.speed.med = median(sword.speed),
   tchests.found.mean = mean(tchests.found)
  ) # End

college.agg # Print result!
```

```
## Source: local data frame [2 x 5]
##
##   college age.med age.min sword.speed.med tchests.found.mean
## 1    CCCC      25       9       0.5485708           6.572998
## 2   JSSFP      33      26       0.5863899           8.495868
```

As you can see, our final object `college.agg` is the aggregated dataframe we want which aggregates all the columns we wanted at the level of the airline carrier. 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 each carrier. 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)`.

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, the following code will restrict data to males

`arrange`

Reorders rows of a dataframe.

`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.

`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.speed.med = median(sword.speed),  # Median sword speed?
              parrots.mean = mean(parrots.lifetime) # # mean parrots?
            ) %>%
            arrange(frequency) # Step 7

college.favpir.agg # Print the result!

## Source: local data frame [12 x 5]
## Groups: college
##
```

```
##    college favorite.pirate frequency sword.speed.med parrots.mean
## 1     CCCC      Blackbeard         1       2.4750347     2.000000
## 2     CCCC      Edward Low         1       1.1246004     2.000000
## 3     CCCC        Anicetus         2       0.7953173     1.500000
## 4     CCCC      Lewis Scot         2       1.0383424     2.000000
## 5     CCCC            Hook         5       0.6542017     2.200000
## 6     CCCC    Jack Sparrow         7       0.5576000     3.142857
## 7    JSSFP      Blackbeard        33       0.7181846     3.333333
## 8    JSSFP      Lewis Scot        34       0.4730343     3.647059
## 9    JSSFP        Anicetus        36       0.5331098     3.583333
## 10   JSSFP            Hook        38       0.4057803     4.394737
## 11   JSSFP      Edward Low        39       0.4551255     3.487179
## 12   JSSFP    Jack Sparrow        81       0.6041815     4.074074
```

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.speed.med` and is the median sword speed 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`

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 wanted to add a column to our happiness dataframe `hsurvey` showing the continent that each person was from. In the dataframe, we only have country information, so how can we add continent info? We can do this by merging the `hsurvey` dataframe with a new dataframe called `continents`. By merging the dataframes, we will add all the information from `contents` to the `hsurvey` dataframe as additional rows. We'll also match the two dataframes with the column country.

```
continents <- data.frame(country = c("USA", "Canada", "Mexico", "India", "Portugal"),
                         continent = c("North America", "North America", "North America", "Asia", "Europe"),
                         gdp = c(53042, 51964.3, 10307.3, 1497.5, 21738.3),
                         stringsAsFactors = F
                         )

hsurvey <- merge(x = hsurvey, # The first dataframe
                 y = continents, # The second dataframe
                 by = "country" # The name of the matching column
                 )

## Error in merge(x = hsurvey, y = continents, by = "country"):  object
’hsurvey’ not found
```

To see if this worked, let's look at the first few rows of hsurvey

```
head(hsurvey)

## Error in head(hsurvey):  object ’hsurvey’ not found
```

As you can see, our new merged dataframe has added the information from continents to the hsurvey dataframe by matching rows with the country column.

Easily recode values in a dataframe with merge()

You can also use the merge() function to quickly recode values in a vector in a dataframe. For example, let's say some drunk pirate accidentally entered the wrong country names for each person in the column country in hsurvey(). Thankfully, when we got him sober enough he could tell us which (incorrect) entry goes to which correct entry. For example: the entry USA should be Canada, Mexico should be India, etc. We can quickly correct this in our original dataframe using merge(). We'll do this in two steps:

1. Create a lookup table that shows the original (incorrect) and the new (correct) values

2. Merge the the original data with the lookup table

Let's create the lookup table called lookup. This will be a dataframe with two columns: country.wrong - the original country values, and country.true the corrected values. Each row in the dataframe will connect the original incorrect value with the new, correct value.

```
lookup <- data.frame("country.wrong" = c("USA", "Canada", "Mexico", "India", "Portugal"),
                     "country.true" = c("Canada", "Mexico", "India", "Portugal", "USA")
                     )
```

Now, let's merge hsurvey with lookup. To do this, we'll need to specify that the matching column in hsurvey is called country, and the matching column in `lookup` is called `country.wrong`. When we merge these two dataframes, R will add the `country.true` column to `hsurvey`

```
hsurvey <- merge(x = hsurvey,
      y = lookup,
      by.x = "country",
      by.y = "country.wrong"
      )

## Error in merge(x = hsurvey, y = lookup, by.x = "country",
by.y = "country.wrong"):  object 'hsurvey' not found
```

Let's see if it worked:

```
head(hsurvey)

## Error in head(hsurvey):  object 'hsurvey' not found
```

10: 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

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.

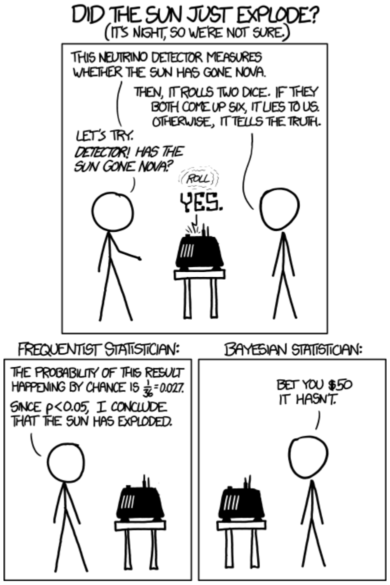

DID THE SUN JUST EXPLODE?
(IT'S NIGHT, SO WE'RE NOT SURE.)
THIS NEUTRINO DETECTOR MEASURES WHETHER THE SUN HAS GONE NOVA.
THEN, IT ROLLS TWO DICE. IF THEY BOTH COME UP SIX, IT LIES TO US. OTHERWISE, IT TELLS THE TRUTH.
LET'S TRY.
DETECTOR! HAS THE SUN GONE NOVA?
ROLL
YES.
FREQUENTIST STATISTICIAN:
THE PROBABILITY OF THIS RESULT HAPPENING BY CHANCE IS 1/36 = 0.027. SINCE p<0.05, I CONCLUDE THAT THE SUN HAS EXPLODED.
BAYESIAN STATISTICIAN:
BET YOU $50 IT HASN'T.

Figure 40: 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`).

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

`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.

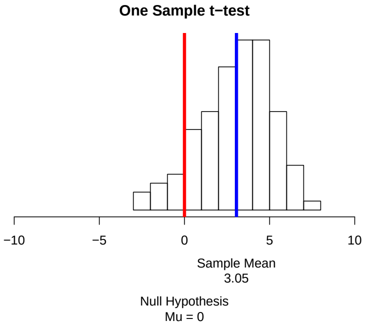
One-Sample t-test

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 on
       )
```

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.

```
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)
```



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 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 = 41.9402, df = 999, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 20
## 95 percent confidence interval:
##  27.28634 28.00166
## sample estimates:
## mean of x
##    27.644
```

Now, you can see the main output of the test. Looks like we got a test statistic of 41.94 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:

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
## 41.94023

test.result$p.value # Show me the p.value

## [1] 1.610482e-222
```

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.

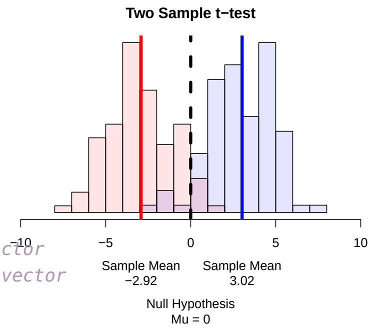
Two-sample t-test

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 = -0.9746, df = 119.921, p-value = 0.3317
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.7489545  0.5951374
## sample estimates:
## mean of x mean of y
##  27.58804  28.16495
```

```
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 -0.97 with a resulting p-value of 0.33. 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 = 0.9746, df = 119.921, p-value = 0.3317
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.5951374  1.7489545
## sample estimates:
##  mean in group no mean in group yes
##          28.16495          27.58804
```

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 = 0.7866, df = 73.284, p-value = 0.434
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.6224917  1.4344088
## sample estimates:
##  mean in group no mean in group yes
##          24.77966          24.37370
```

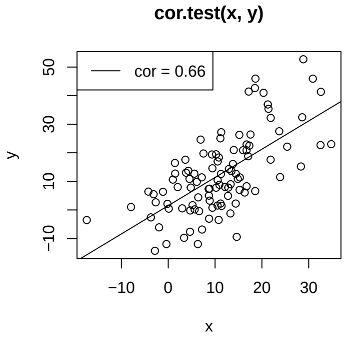
Looks like we still don't find a significant difference in age between headband and no-headband wearers, even just for pirates who went to Captain Chunk's Canon Crew.

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()`

`x, y`

Two numeric data vectors of the same length

`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.lifetime
                        )

test.result

##
##  Pearson's product-moment correlation
##
## data:  pirates$age and pirates$parrots.lifetime
## t = 9.1339, df = 998, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.2195387 0.3339937
## sample estimates:
##       cor
## 0.2777516
```

Looks like we have a positive correlation of 0.28! 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] 0.2195387 0.3339937
## 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`

```
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 = 4.253, df = 3, p-value = 0.2354
```

Looks like we got a test-statistic of 4.25 and a p-value of 0.24. 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      25     546    33       33
##           JSSFP     14     296    29       24
```

Cool. It looks like R stores a table of the observed frequencies and the expected frequencies under the null-hypothesis. Thanks R!

11: Regression and ANOVA

Chapter Goals

1. Learn about regression

2. ANOVA

The Linear Model

In this chapter, we will go over the basics of applying the general linear model in R.

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 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:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ...\beta_n x_n$$

where the x values represent the predictors, while the beta values represent weights. 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).

To use the 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 `subset = age > 50 sex == "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. 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.

```
shipauction <- read.table("http://nathanieldphillips.com/wp-content/uploads/2015/06/shipauction.txt", sep = "\t")
```

First, 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)
```

Because we are including all columns in the `shipauction` dataframe as independent variables, we can also write our regression command as:

```
auction.lm <- lm(price ~ ., data = shipauction)
```

The results will be exactly the same. Using this format will save you a lot of typing if you are running regression analyses on dataframes with many independent variables!

When we call the `lm()` function and assign it to an object, R won't spit out any output. 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
## 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.

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)`):

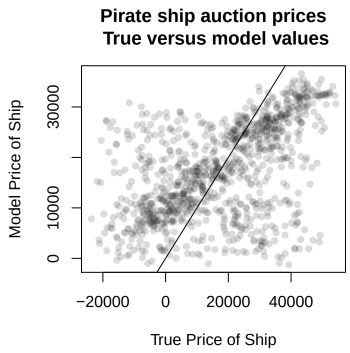
- `coefficients`: A vector of the estimated beta values corresponding to the independent variables (and the intercept)
- `fitted.values`: A vector of the fitted values for all test cases. This is the fitted linear model's prediction for the dependent variable (in our case, price) for each ship.
- `residuals`: 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

[4] Note: The text `Pr(>|t|)` is the p-value.

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 vers
     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*

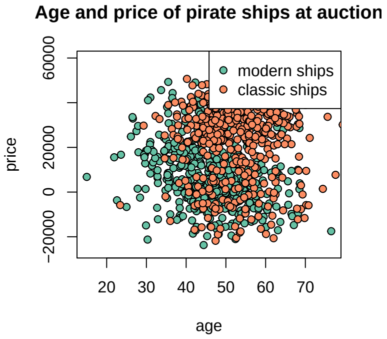
An interaction in a linear model means that the relationship between one independent variable on the dependent variable depends on the level of another independent variable. For example, let's see if the relationship between `age` and `price` depends on the level of `style`. To keep things simple, we'll 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 relationship, 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:

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.

```
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")
```



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:

$$p(y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_n x_n)}}$$

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

```
# 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)
```

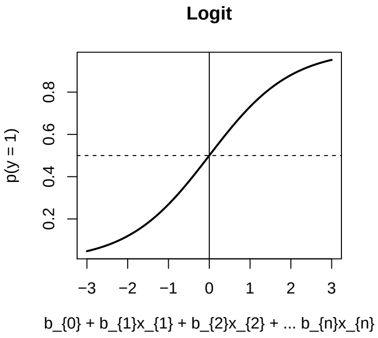


Figure 41: The logit function used in binary logistic regression

to be classic ships than modern ships. However, before we run the 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:

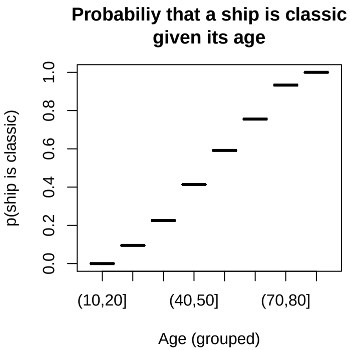
```
shipauction$style.num[shipauction$style == "modern"] <- 0
shipauction$style.num[shipauction$style == "classic"] <- 1
```

```
# Step 4: Run the model then print the result
age.style.glm <- glm(style.num ~ age,
              data = shipauction, family = "binomial")

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 Margin Figure XX: 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")
```

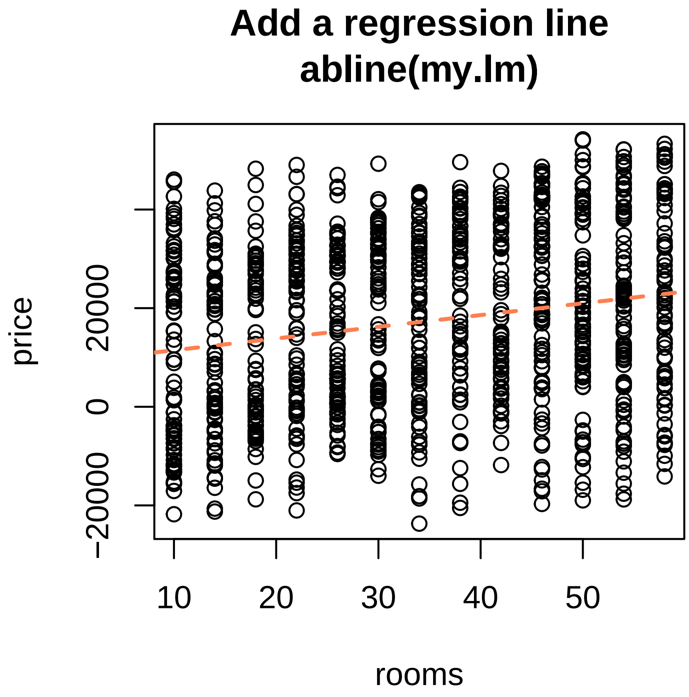


Additional Tips

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(shipauction$rooms, shipauction$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 = 2, lwd = 2, col = "coral")
```



Calculating regression lines on subsets of data

To run a linear model on a subset of data in a dataframe, use the `subset` argument. For example, to calculate a regression model on the pirates dataset, but only for pirates whose sex is Female and who are less than 30 years old, we can do the following:

```
model <- lm(age ~ college + parrots.lifetime,
            data = pirates,
            subset = sex == "female" & age < 30
            )
```

Predicting new data from a model

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 create a regression model that tries to predict the price of a ship based just on

12: Writing your own functions

Why would you want to write your own function?

To repeat the same code several times in the same session, or in different sessions

To create code that can change based on some input - A plotting function that can add elements you specify

The basic structure of a function

What is a function? A function is simply an object that (usually) takes some input, performs some action, and then (usually) returns some output. This might sound complicated, but you've been using functions pre-defined in R throughout this book. For example, the function `mean()` takes a numeric vector as an argument, and then returns the arithmetic mean of that vector as a single scaler value.

All functions have the following basic structure. First you define the name of the function, and assign a function call with `function(x, y, ...)` to that name. Next, you specify the inputs to the function `x, y, ...` within the function call. Finally, you specify what the function should do in curly braces:

```
my.fun <- function(inputs) {function code}
```

Here, `my.fun` is the name of the function you are creating, `inputs` are the inputs to the function, and `code` is any R code that you want the function to run. If you want the function to return some value to the user, you'll include the output in the `return()` function.

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 [6]:

```
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

}
```

When you create the function, nothing will happen - 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
##
## }
```

Using multiple inputs and defaults

You can create functions with as many inputs as you'd like (even 0!). For example, we could create a simple function called my.fun that takes three arguments a, b, c and returns (a + b) * c

```
my.fun <- function(a, b, c) {

  output <- (a + b) * c

  return(output)
}
```

Now let's test our new function with a few different values for the inputs a, b and c.

```
my.fun(a = 1, b = 4, c = 1)

## [1] 5
```

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:

```
my.fun.vec <- function(input) {

  a <- input[1]
  b <- input[2]
  c <- input[3]

  output <- (a + b) * c

  return(output)

}
```

[6] The return() function is a special function that only use when you are writing a function. Anytime you want a function to return some value to the user, you need to specify this using return() by putting whatever object you want the function to return in the parentheses

```
my.fun(a = 0, b = 0, c = 5)

## [1] 0
```

Now, let's add default values to the inputs. What is a default value? A default value is a value that R will assign to an input if the user does not specify it explicitly when calling the function. For example, if a user runs `my.fun()` but does not specify a value for c, we can create a default value that R will use. To add a default value to an input, just add `= X`, where X is the default value. Let's add some default values to `my.fun`, so that a will default to 5, b will default to 10, and c will default to 2:

```
my.fun <- function(a = 5, b = 10, c = 2) {

  output <- (a + b) * c

  return(output)
}
```

Now, if a user does not specify a value for a, b, or c, the function will set the input values to 5, 10 and 2 respectively.

Now, let's see what happens when we run `my.fun()` without specifying specific values:

```
my.fun(a = 100, b = 0, c = 2)

## [1] 200

my.fun(a = 100, b = 0)  # c will default to 2

## [1] 200

my.fun()  # All values will be their default

## [1] 30
```

Creating multiple coinflip() functions

Now let's create an extended example of creating a function that simulates the flip of a coin. We'll call this function `coinflip`.

For the first version of the function (`coinflip.1`), we'll create it with 0 arguments. When the function runs, it will simply return either "Heads" or "Tails" with equal probability

[7]

[7] If you're not familiar with the `sample()` function, look at Chapter 3 or look at the help page by executing `?sample`

```
coinflip.1 <- function () {

  output <- sample(x = c("Heads", "Tails"),
                   size = 1,
                   prob = c(.5, .5),
                   replace = T
                   )

  return(output)

}
```

Because our function `coinflip.1()` has no inputs, we can call it by just running `coinflip.1()`:

```
coinflip.1()

## [1] "Heads"
```

As you can see, our function has returned a single string scalar representing the outcome of a single coin flip. Because the outcome is randomly generated, you can get different values each time you run it.

Now, let's make the `coinflip` function a bit more complicated by adding additional inputs. We'll create a new version of the function called `coinflip.2()`. Here, we'll add an input called `n.flips` that indicates the number of flips of the coin we want, and `head.p`, the probability of flipping heads on any coin toss. Once we've defined these as inputs, we can then refer to them in the main function code (by putting them into the `sample()` function).

```
coinflip.2 <- function (n.flips, head.p) {

  output <- sample(x = c("Heads", "Tails"),
                   size = n.flips, # Now the size argument is n.flips
                   prob = c(head.p, 1 - head.p), # Probability of "Heads" is head.p
                   replace = T
                   )

  return(output)

}
```

Let's test our new function by simulating 20 flips of a coin where the probability of heads is .90. If the function works correctly, we should get a string vector of length 20, where approximately 90% of the values are "Heads".

```
coinflip.2(n.flips = 20, head.p = .9)

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

Looks like we did indeed get a vector of length 20 with mostly heads.

Now, let's add default values to the inputs. Let's create `coinflip.3` that includes a default value of 1 for `n.flips`, and a default value of 0.50 for `head.p`:

```
coinflip.3 <- function (n.flips = 1, head.p = .5) {

  output <- sample(x = c("Heads", "Tails"),
                   size = n.flips,
                   prob = c(head.p, 1 - head.p),
                   replace = T
                   )

  return(output)

}
```

Now, let's try running `coinflip.3` with a different number of inputs. In each of these function calls, if I do not specify one of the two inputs, R will use the default values we specified earlier (that is, `n.flips = 1` and `head.p = .5`)

```
coinflip.3(n.flips = 5, head.p = .5)  # Specify all inputs

## [1] "Heads" "Tails" "Heads" "Heads" "Heads"

coinflip.3(head.p = .1)  # Just specify head.p

## [1] "Heads"

coinflip.3(n.flips = 10) # Just specify n.flips

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

coinflip.3()  # Don't specify anything!

## [1] "Heads"
```

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 if the logical test is TRUE. If the logical test is FALSE, R will completely ignore all the code in the curly braces.

In the following code, R will test if the object a in the parentheses is greater than 0. If it is, it will print "This value is positive"

```
a <- 3
if(a > 0) {print("This value is positive")}

## [1] "This value is positive"
```

If a is less than 0, R won't print anything because it will completely ignore everything in the curly braces

```
a <- -10
if(a > 0) {print("This value is positive")}
# Nothing happens because a is not greater than 0!
```

For example, let's create a function called `do.this()` that takes a numeric vector `x` as an argument, and then includes an additional string argument called `what` that tells the function what to do. In the function, we'll do different things depending on the value of `what`. If the value of `what` is the string "sum", the function will calculate the sum of all the values in x. If the value of `what` is the string "mean", the function will calculate the mean of all the values in x. Finally, if the value of `what` is "joke", the function will return a really funny joke:

```
do.this <- function(x, what) {

  if (what == "sum") {output <- sum(x)}
  if (what == "mean") {output <- mean(x)}
  if (what == "joke") {output <- "I'm a pirate, not your joke monkey"}

  return(output)

}
```

Let's test the function using different values of `do.what`

You can put anything into an if() statement

You can put any R code you want in a `if()` function (not just `print()`). For example, let's create an if command that checks if a numeric scaler is negative. If it is negative, R will do one set of things, if a is positive, R will do another set of things.

```
a <- -50  # set a to -50

if(a < 0) { # Code when a is negative

 print("A is negative! I love negative numbers! Here are some more")
 sample(seq(-100, 0, 1), size = 5) # Print some random negative numbers

} # End of things to do if a is negative


## [1] "A is negative! I love negative numbers! Here are some more"
## [1] -12  -6  -7 -29 -33


if(a > 0) { # Code when a is positive

  print("I hate positive numbers! Let's make a negative!!")
  a <- -1 * a

  print("ok! Now a is negative")
  a

} # End of things to do if a is positive
```

In this example, I set a to -50, so R only executed the code for the case where a < 0. If you change the value of a to a positive number, then R will execute the code for the if statement when a > 0.

```
do.this(x = c(1, 5, 3, 2, 1, 2), what = "sum")

## [1] 14

do.this(x = c(1, 5, 3, 2, 1, 2), what = "mean")

## [1] 2.333333

do.this(x = c(1, 5, 3, 2, 1, 2), what = "joke")

## [1] "I'm a pirate, not your joke monkey"
```

As you can see, R has changed the output depending on what the user enters for the `do.what` argument.

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 `do.this` function, we only considered three possible inputs (`"sum", "mean", "joke"`) for the `what` argument. If a user enters a different value for the `what` argument, the function will fail and return some obtuse errors. Let’s see what happens when we enter `what = "product"` as an argument:

```
do.this(x = 1:10, what = "product")

## Error in do.this(x = 1:10, what = "product"):  object
’output’ not found
```

R returned an error because when `what` is “product”, the function never defines the `output` object and thus returns an error. The problem with this though is the error message is quite vague, and without knowing the function in detail the user might not know what he/she did wrong. 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 `do.this` 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:

```
do.this <- function(x, what) {

  valid.input <- what %in% c("sum", "mean", "joke")

  if(valid.input == TRUE) {  # Begin TRUE valid input section

  if (what == "sum") {output <- sum(x)}
  if (what == "mean") {output <- mean(x)}
  if (what == "joke") {output <- "I'm a pirate, not your joke monkey"}

  } # Close TRUE valid input section

  if(valid.input == FALSE) { # Begin FALSE valid input section

    output <- "Your input for what is invalid. Please enter sum, mean, or joke. Try joke, it's funny"
  } # Close FALSE valid input section

  return(output)

}
```

Let's try it out. We'll execute the `do.this()` function with an invalid value of `what`

```
do.this(x = 5, what = "product")

## [1] "Your input for what is invalid. Please enter sum, mean, or joke. Try joke, it's funny"

do.this(x = 1, what = "joke")

## [1] "I'm a pirate, not your joke monkey"
```

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 (in the curly braces).

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 what 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[8]. The ... input tells R that the user might add additional inputs that should be used later in the function.

Here's a quick example, let's create a function called `hist.advanced` that plots a histogram with some optional additional arguments passed on with ...

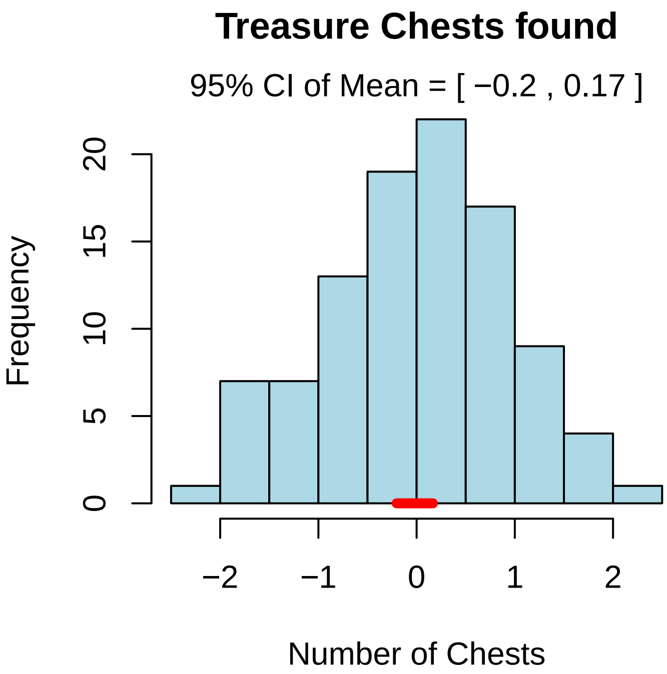
```
hist.advanced <- function(x, add.ci = T, ...) {

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")
```

[8] 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.

```
  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()`:

```
hist.advanced(x = rnorm(100), add.ci = T,
              main = "Treasure Chests found", xlab = "Number of Chests",
              col = "lightblue")
```



As you can see, R has passed our optional plotting arguments down to the main hist() function in the function code.

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.

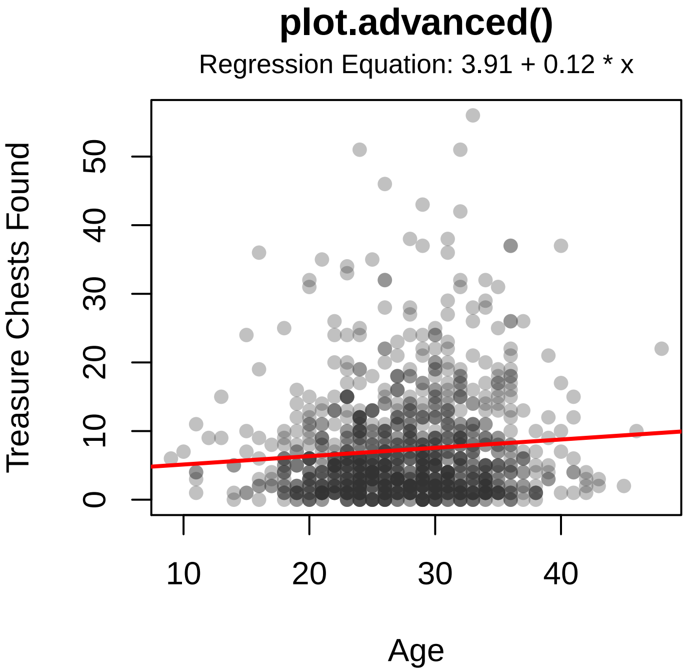
- add.regression: A logical value indicating whether or not to add a linear regression line
- sig.line.col: The color of the regression line if the slope is significant.
- nonsig.line.col: The color of the regression line if the slope is NOT significant.
- p.threshold: A numeric scaler indicating the 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 complicated, so don't worry if you don't understand everything it right away.

```
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 turn on some of the additional elements, and include optional titles for the plot:

```
plot.advanced(pirates$age, pirates$tchests.found,
              add.regression = TRUE,
              add.modeltext = TRUE,
              p.threshold = .05,
              main = "plot.advanced()",
              xlab = "Age", ylab = "Treasure Chests Found",
              pch = 16, col = gray(.2, .3))
```


plot.advanced()
Regression Equation: 3.91 + 0.12 * x
Treasure Chests Found
0
10
20
30
40
50
10
20
30
40
Age

13: Loops and Simulations

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. 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. In R, the format of a for-loop is:

```
for(INDEX in INDEX.VALUES) {LOOP.CODE}
```

For example, the following is a for-loop that prints the integers from 1 to 10:

```
for (i in 1:10) {print(i)}

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

There are three key aspects of loops:

1. index: The object that is changing according to the index values. In the previous example, the index is `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 mean something. For example, if you are doing a loop over participants in a study, you can call the index `participant.i`.

2. index values: 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")`.

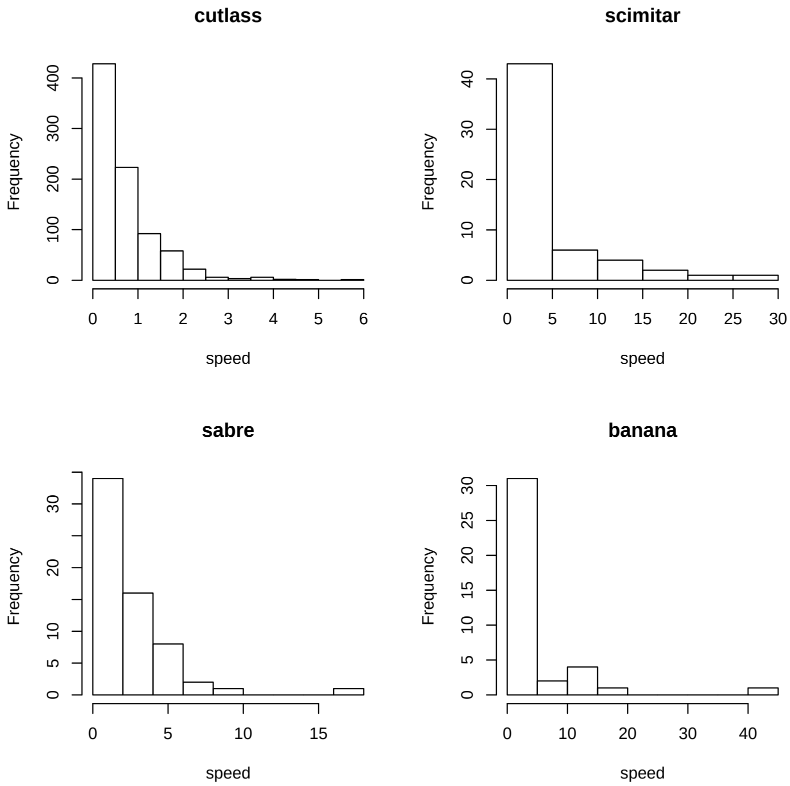
3. loop code: The code that will be executed for all index values. In the previous example, the loop code is `print(i)`. You can write any R code you’d like in a loop - from plotting, to analyses.

Let’s do some simple for-loops on our pirates dataset.

Creating multiple plots with a loop

Let’s say we wanted to create a histogram of the sword speeds for each sword type. Instead of writing the `hist()` function several times, we’ll create a loop.

```
par(mfrow = c(2, 2))  # Set up a 2 x 2 plotting space

for (sword.type.i in c("cutlass", "scimitar", "sabre", "banana")) {

  # subset data for current index value
  data.temp <- subset(pirates,
                      sword.type == sword.type.i)

  # Plot histogram of temporary data
  hist(data.temp$sword.speed,
       main = sword.type.i, # Set plot title to index value
       xlab = "speed")

}
```



Here's how the loop above works. First, I set up a 2 x 2 plotting space with `par(mfrow())`. Next, I started the loop by defining the index as `swordtype.i`, and the index values as `c(cutlass, scimitar, sabre, banana)`[9]. 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, for those cases where the column `sword.type` is equal to the index value `sword.type.i`. Next, I created a histogram from the `sword.speed` column from `data.temp` where the main title of the histogram is equal to `sword.type.i`

Assigning values to an object by combining a loop and assignment

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.

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

[9] I could have made this code a bit simpler by defining the index values as `unique(pirates$sword.type)`. This would save me from having to type all possible values of the sword type.

to ten.

The first thing we need to do is create an vector called squares that is filled with NA values. When we write the loop, we'll overwrite these NA values by assigning new values. Anytime you want to save values in a loop, you need to make sure an object (like a matrix or dataframe) exists to store the values before you start the loop. Personally, I like to create vectors, matrices, or dataframes that are filled with NA values. Then, once the loop is completed, I can check to see if any NA values are left in the object. If they are, then something probably went wrong with the loop.

```
squares <- rep(NA, 10)
```

Now we can do our loop. We'll loop over index values from 1 to 10. For each index value, we'll calculate the square of that 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.

In fact, we could have easily done the same thing with:

```
squares <- (1:10) ^ 2
```

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:

```
# 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 <- as.data.frame(matrix(NA, nrow = n.ages, ncol = 4))
names(parrot.ci.df) <- c("age", "mean", "lb", "ub")

# Print the result
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
```

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.

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$parrots.lifetime) # Get the sample mean
  ci.i <- t.test(data.temp$parrots.lifetime) # 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]
```

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`

```
} # Close loop
```

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 1.969697 5.918269 32
## 2   21 1.742857 5.472772 34
## 3   22 1.350000 6.072519 39
## 4   23 2.649123 6.557254 56
## 5   24 2.080000 8.195202 74
## 6   25 3.075472 7.418234 52
## 7   26 2.363636 7.671259 65
## 8   27 2.285714 8.302718 55
## 9   28 2.416667 7.860791 59
## 10  29 3.072464 6.818705 68
## 11  30 3.014085 8.095216 70
```

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         32 32.00000 2.926639
## 2    CCCC   male         24 23.50627 4.189522
## 3   JSSFP female         34 33.77821 3.469937
## 4    CCCC female         26 26.02857 3.422294
```

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 of lists....).

To create a list in R, use the `list()` function. Let's create a list that contains 3 vectors where each vector is a random sample from a normal distribution. We'll have the first element have 10 samples, the second will have 5, and the third will have 15.

```
number.list <- list("first" = rnorm(n = 10),
     "second" = rnorm(n = 5),
     "third" = rnorm(n = 15)
)

number.list

## $first
##  [1]  1.28044230 -1.38974668 -1.78883875  0.07580274 -1.28073515
##  [6] -0.47139643 -0.09711147  0.69950510 -1.88002510 -1.98666366
```

```
##
## $second
## [1] -3.4646438 -1.7647492  0.7151397  0.7276283 -0.5690954
##
## $third
##  [1] -1.47062232 -0.93942472 -0.94036811 -0.02026055  0.32091253
##  [6]  1.29690583  0.40908405 -0.29338873  0.15593591  0.88325099
## [11] -0.19105124  0.43316408  2.22196760  0.66567183 -0.34316132
```

To index a list, use double brackets [[]] or $ if the list has names. For example, to get the first element of a list named `number.list`, we'd use `number.ls[[1]]`:

```
number.list[[1]]

##  [1]  1.28044230 -1.38974668 -1.78883875  0.07580274 -1.28073515
##  [6] -0.47139643 -0.09711147  0.69950510 -1.88002510 -1.98666366

number.list$second

## [1] -3.4646438 -1.7647492  0.7151397  0.7276283 -0.5690954
```

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] -1.027127
##
## [[2]]
## [1] -0.5933634  1.4533802
##
## [[3]]
## [1] -1.0444339 -0.5516022 -0.7425344
##
## [[4]]
## [1] -0.9008107  0.1950480  0.5663279  1.3993161
##
## [[5]]
## [1]  0.13906197 -0.26122046 -0.02065071  0.06000143  0.23857757
```

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] -1.02712746 -0.59336344  1.45338018 -1.04443387 -0.55160215
##  [6] -0.74253437 -0.90081071  0.19504805  0.56632791  1.39931610
## [11]  0.13906197 -0.26122046 -0.02065071  0.06000143  0.23857757
```

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.

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[10]. Once you execute this, R will prepare those slaves in the background.

[10] 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 3.1.3 (2015-03-09)

## snowfall 1.84-6 initialized (using snow 0.3-13):  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()`[11] 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:

```
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
## 33.3168 24.4113
```

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

[11] In addition to `sfSapply()`, you can also use `sfLapply()`, which returns stores the results as a list

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 sim
                                   )
```

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.

A worked example: What is a p-value anyway?!

P-values are one of the most confusing aspects of statistics. If you ask 10 different R pirates what a p-value means, you'll probably get 10 different answers. It's not because R pirates are stupid, it's because p-values are inherently confusing as hell. Let's use some simulations to see what they really mean.

For this simulation, I'll draw random samples of size 10 from Normal distributions with some specified mean. I'll let the true population mean range from 0 (where the null hypothesis is true) to 10 in steps of 1. I'll fix the population standard deviation to be 1.

I'll start by creating the design matrix. This matrix will have one two varying dimensions: population.mean - the true mean of the population, which will range from 0 to 10, and simulation - the simulation number for the current parameter settings, which will range from 1 to 1,000. In addition, I'll specify the population standard deviation (population.sd) to always be 1.

```
design.matrix <- expand.grid("population.mean" = c(0:10),
                             "population.sd" = 1,
                             "simulation" = 1:1000,
                             "p.value" = NA
                             )
```

Let's take a look at the design.matrix:

```
set.seed(100) # Fix the random seed
dim(design.matrix) # Get dimensions of design.matrix

## [1] 11000     4

design.matrix[sample(nrow(design.matrix), 10),] # Look at 10 random entries

##      population.mean population.sd simulation p.value
## 3386               8             1        308      NA
## 2835               7             1        258      NA
## 6075               2             1        553      NA
## 621                4             1         57      NA
## 5153               4             1        469      NA
## 5320               6             1        484      NA
## 8932              10             1        812      NA
## 4071               0             1        371      NA
## 6008               1             1        547      NA
## 1872               1             1        171      NA
```

Now, let's add a column indicating whether the null hypothesis really is true or false:

```
design.matrix$null.hypothesis <- design.matrix$population.mean == 0
```

Ok, now let's do our simulation. I'll loop through each row in the design matrix, extract the population mean and standard deviation, draw a sample of size 10 from that distribution, then conduct a 1 sample t.test. I'll then extract the p-value from the t-test and assign it to the corresponding entry in the p.value column of the design.matrix.

```
for (row.i in 1:nrow(design.matrix)) {

  # Get population parameters
  mean.i <- design.matrix$population.mean[row.i]
  sd.i <- design.matrix$population.sd[row.i]

  # Draw a sample with parameters
  sample.i <- rnorm(10, mean = mean.i, sd = sd.i)

  # Conduct t.test
  test.result <- t.test(sample.i)

  # Get p.value
  p.value.i <- test.result$p.value

  # Write p.value to design.matrix
  design.matrix$p.value[row.i] <- p.value.i

}
```

Now, let's calculate a new column indicating whether or not a p.value is significant at the .05 level:

```
design.matrix$sig.05 <- design.matrix$p.value < .05
```

Now let's aggregate the results. First, let's aggregate the data at the level of the true population mean. In other words, for each population mean, what is the probability of getting a significant test result?

```
agg.popmean <- aggregate(sig.05 ~ population.mean, data = design.matrix, FUN = mean)
```

Now, let's aggregate the data at the level of the significance and see how often the null hypothesis is really true. This will tell us, if we have a significant test result, what is the probability that the null hypothesis is really true?

```
agg.sig <- aggregate(null.hypothesis ~ sig.05, data = design.matrix, FUN = mean)
```

```
agg.1 <- aggregate(sig.05 ~ null.hypothesis, data = design.matrix, FUN = mean)
agg.2 <- aggregate(null.hypothesis ~ sig.05, data = design.matrix, FUN = mean,
                   subset = population.mean %in% c(0, 1))
```

14: Bayesian Inference (Coming Soon!)

What are Bayesian statistics?

Bayesian one and two sample tests

Bayesian general linear model

15: Model fitting (Coming Soon!)

What is a model?

What is a loss function?

Minimizing loss functions with optimization routines

A worked example: Prospect Theory

16: Writing and sharing your work (Coming Soon!)

RMarkdown

Shiny

Sweave (R and Latex)

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
     )
```

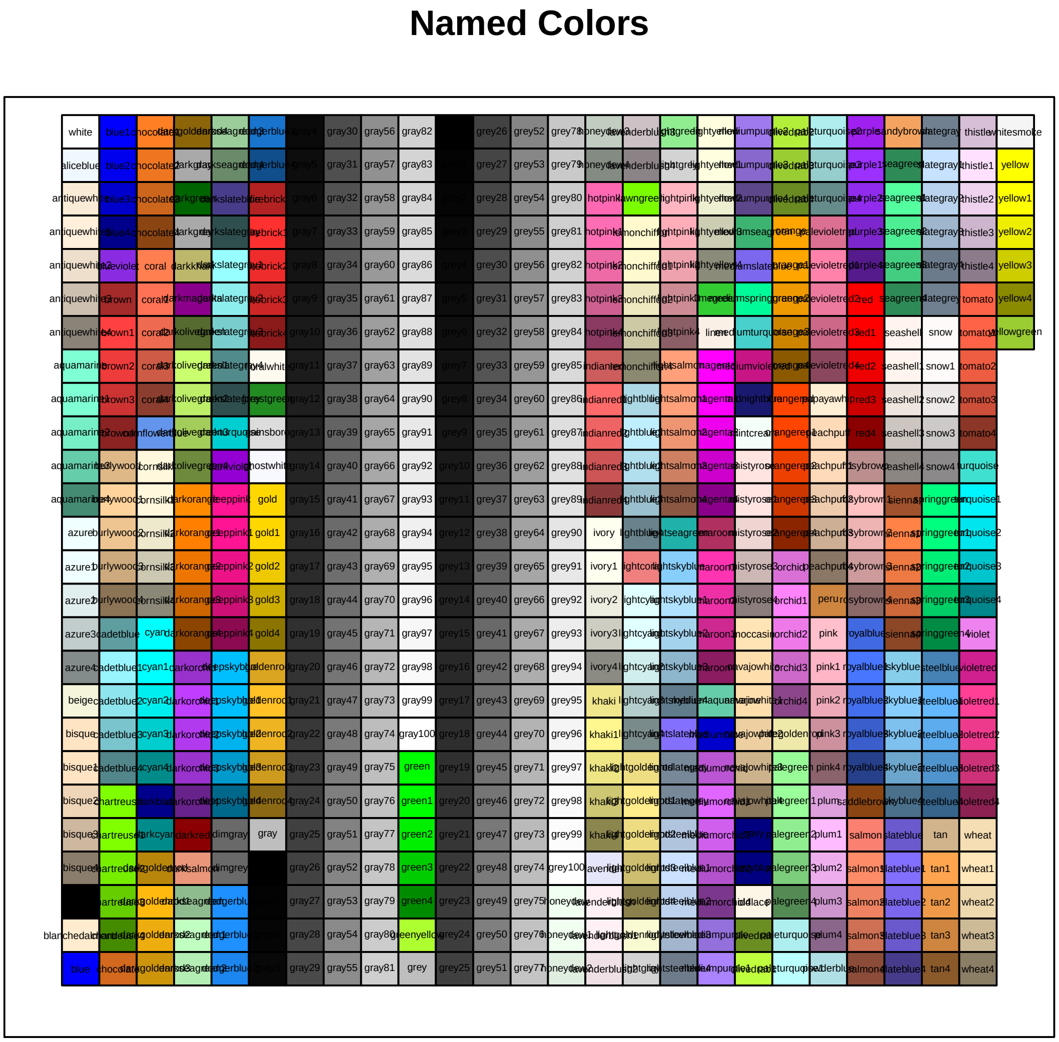


Figure 42: The colors stored in `colors()`.

Index