Chapter 4

The Basics

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

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

4.1 The command-line (Console)

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

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

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

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

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

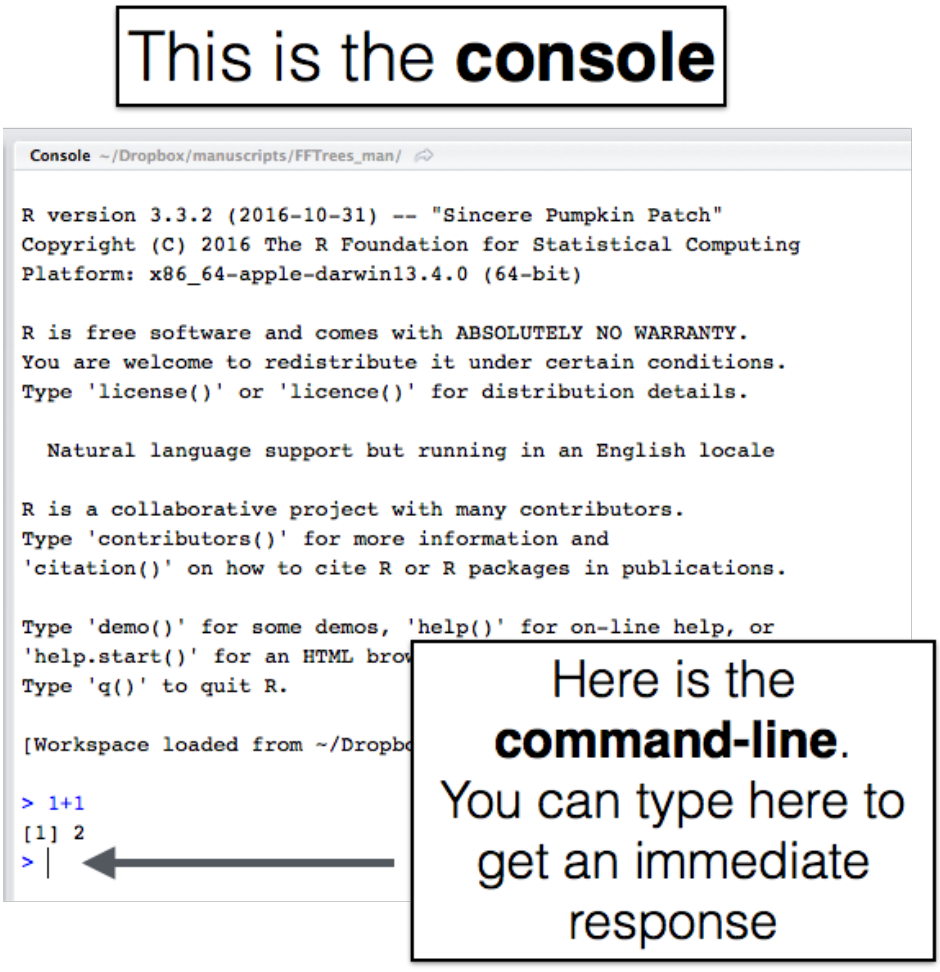


Figure 4.3: You can always type code directly into the command line to get an immediate response.

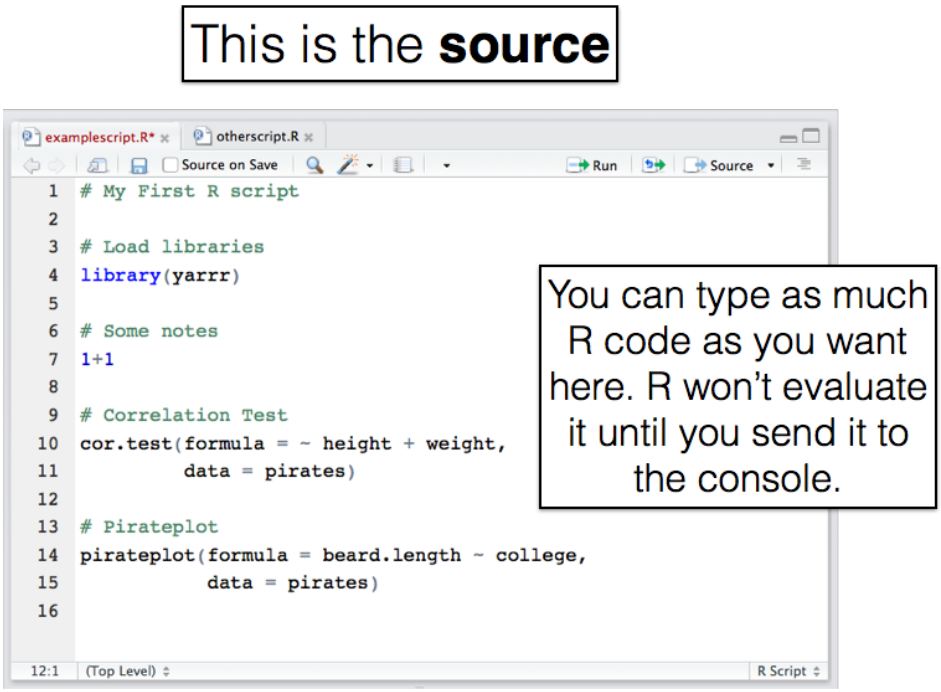


Figure 4.4: Here's how a new script looks in the editor window on RStudio. The code you type won't be executed until you send it to the console.

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

4.2 Writing R scripts in an editor

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

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

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

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

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

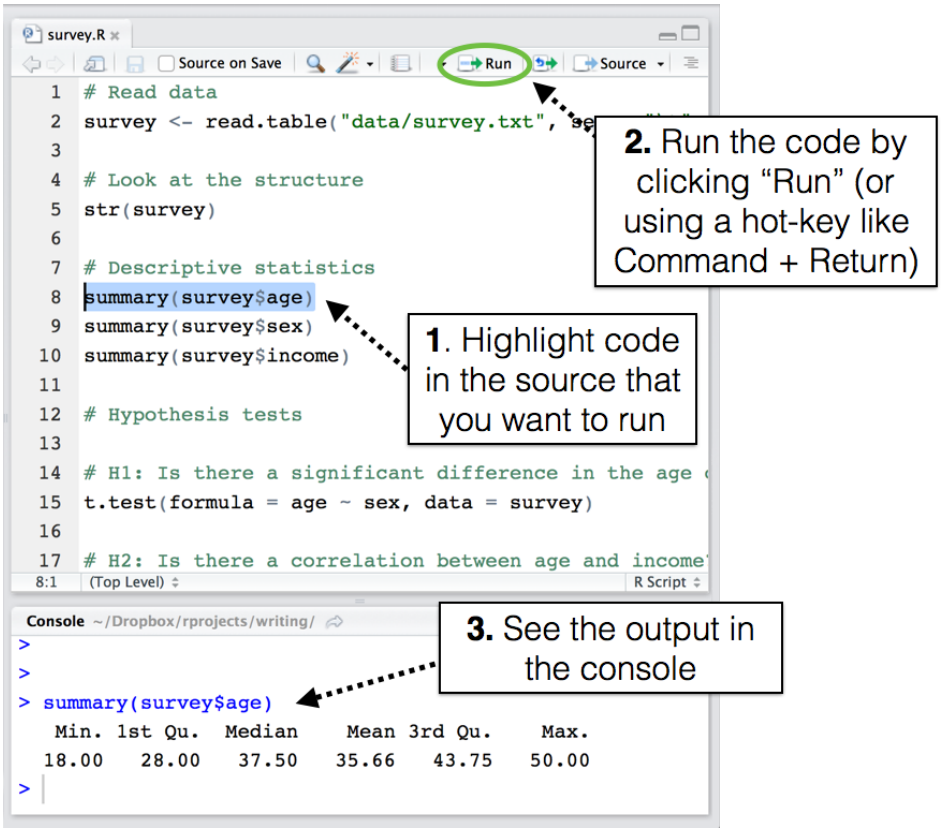


Figure 4.5: To evaluate code from the source, highlight it and run it.

4.2.1 Send code from an source to the console

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

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

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

4.3 A brief style guide: Commenting and spacing

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

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

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

4.3.1 Commenting code with the # (pound) sign

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

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

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

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

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

# Step 3: Calculations

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

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

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

4.3.2 Spacing

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

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

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

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

```
# Some meaningless calculations. Not important

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

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

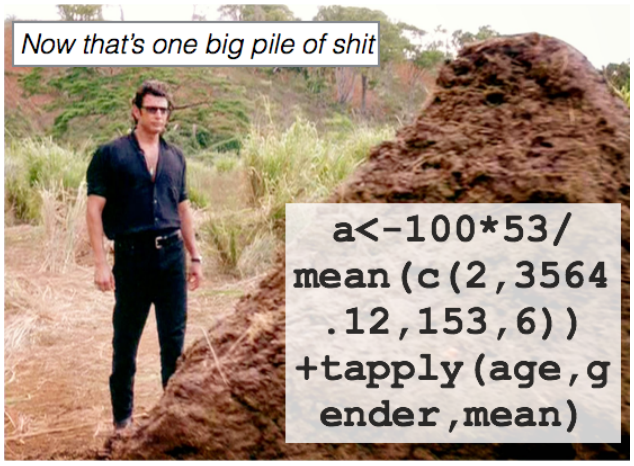


Figure 4.8: Don't make your code look like what a sick Triceratops with diarrhea left behind for Jeff Goldblum.

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

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

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

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

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

4.4 Objects and functions

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

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

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

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

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

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

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

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

4.4.1 Numbers versus characters

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

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

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

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

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

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

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

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

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

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

4.4.2 Creating new objects with <-

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

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

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

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

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

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

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

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

```
b
```

Error: object 'b' not found

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

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

```
a <- 1
b <- 100

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

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

# What is c?
```

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

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

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

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

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

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

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

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

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

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

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

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

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

4.4.3 How to name objects

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

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

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

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

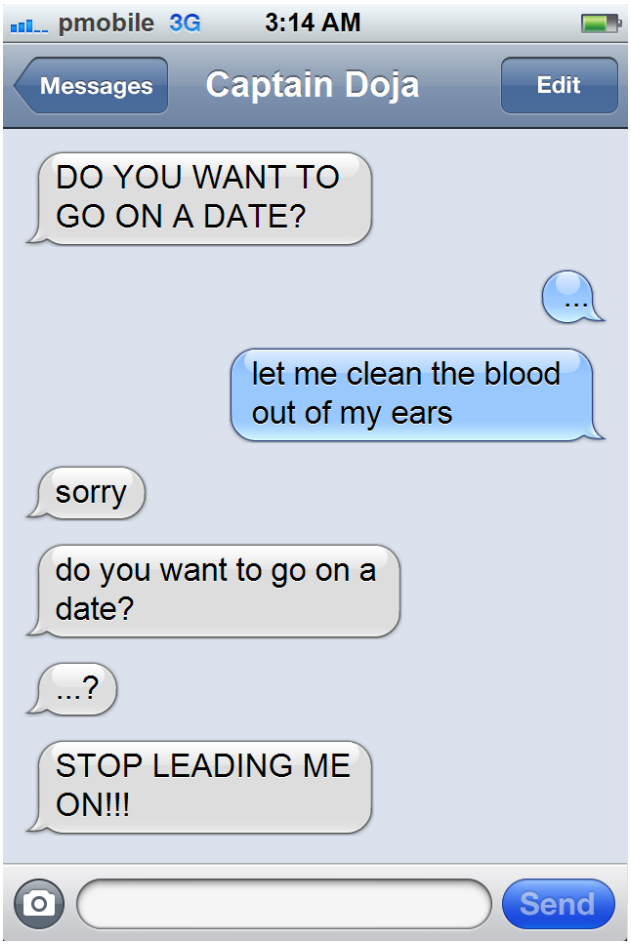


Figure 4.9: Like a text message, you should probably watch your use of capitalization in R.

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

Error: unexpected symbol

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

4.4.3.1 R is case-sensitive!

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

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

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

4.4.4 Example: Pirates of The Caribbean

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

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

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

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

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

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

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

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

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

4.5 Test your R might!

1. Create a new R script. Using comments, write your name, the date, and "Testing my Chapter 2 R Might" at the top of the script. Write your answers to the rest of these exercises on this script, and be sure to copy and paste the original questions using comments! Your script should **only** contain valid R code and comments.
2. Which (if any) of the following objects names is/are invalid?

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

3. 2015 was a good year for pirate booty - your ship collected 100,800 gold coins. Create an object called `gold.in.2015` and assign the correct value to it.
4. Oops, during the last inspection we discovered that one of your pirates Skippy McGee hid 800 gold coins in his underwear. Go ahead and add those gold coins to the object `gold.in.2015`. Next, create an object called `plank.list` with the name of the pirate thief.
5. Look at the code below. What will R return after the third line? Make a prediction, then test the code yourself.

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

Chapter 5

Scalars and vectors

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

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

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

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

5.1 Scalars

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

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

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

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

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

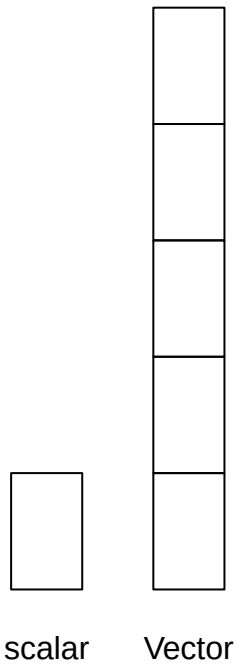


Figure 5.1: Visual depiction of a scalar and vector. Deep shit. Wait until we get to matrices - you're going to lose it.

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

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

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

5.2 Vectors

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

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

Table 5.1: Functions to create vectors.

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

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

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

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

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

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

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

5.2.1 a:b

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

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

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

5.2.2 seq()

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

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

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

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

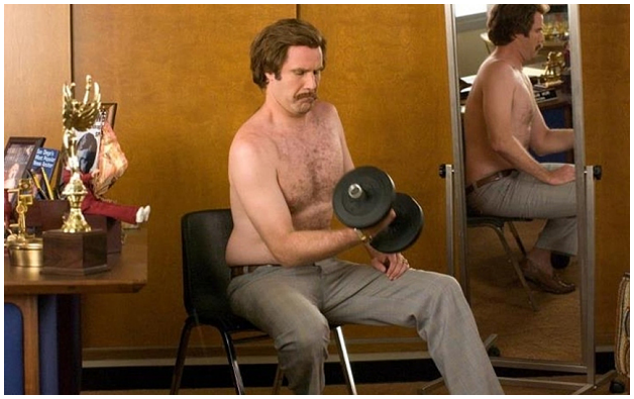


Figure 5.3: Not a good depiction of a rep in R.

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

5.2.3 rep()

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

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

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

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

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

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

Warning! Vectors contain either numbers or characters, not both

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

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

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

5.3 Generating random data

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

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

5.3.1 sample()

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

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

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

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

5.3.1.1 replace = TRUE

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

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

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

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

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

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

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

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

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

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

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

5.3.1.2 Ex: Simulating coin flips

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

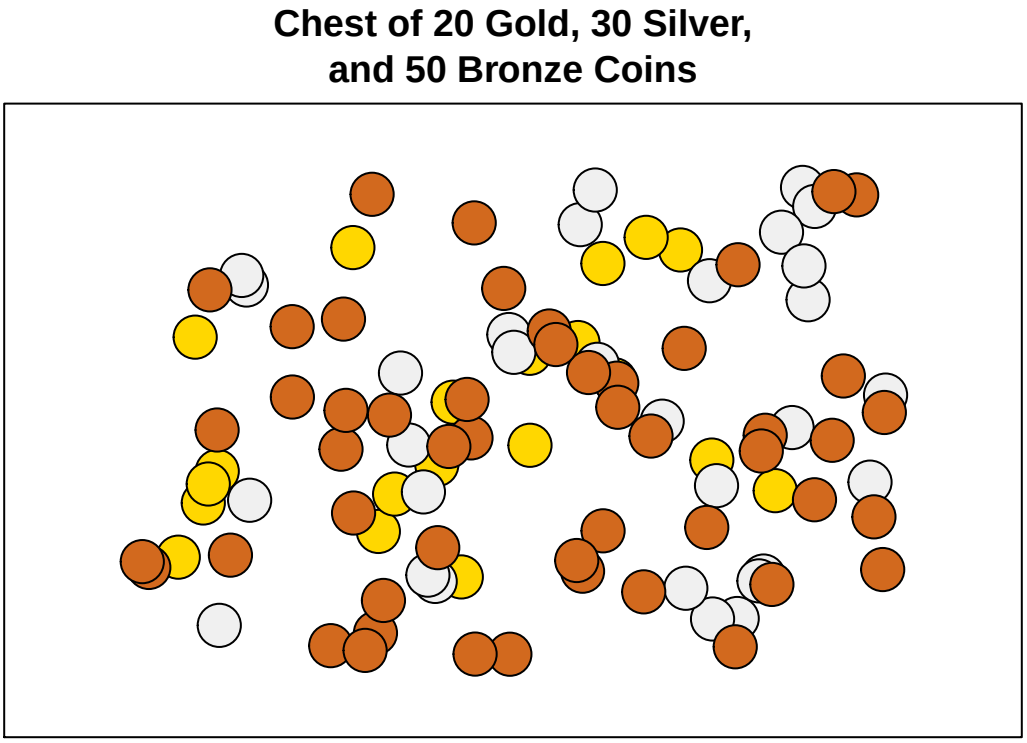
```
sample(x = c("H", "T"), # The possible values of the coin
       size = 10,  # 10 flips
       replace = TRUE) # Sampling with replacement
##  [1] "H" "H" "T" "T" "H" "T" "H" "T" "H" "H"
```

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

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

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

5.3.1.3 Ex: Coins from a chest



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

```
# Create chest with the 100 coins

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

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

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

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

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

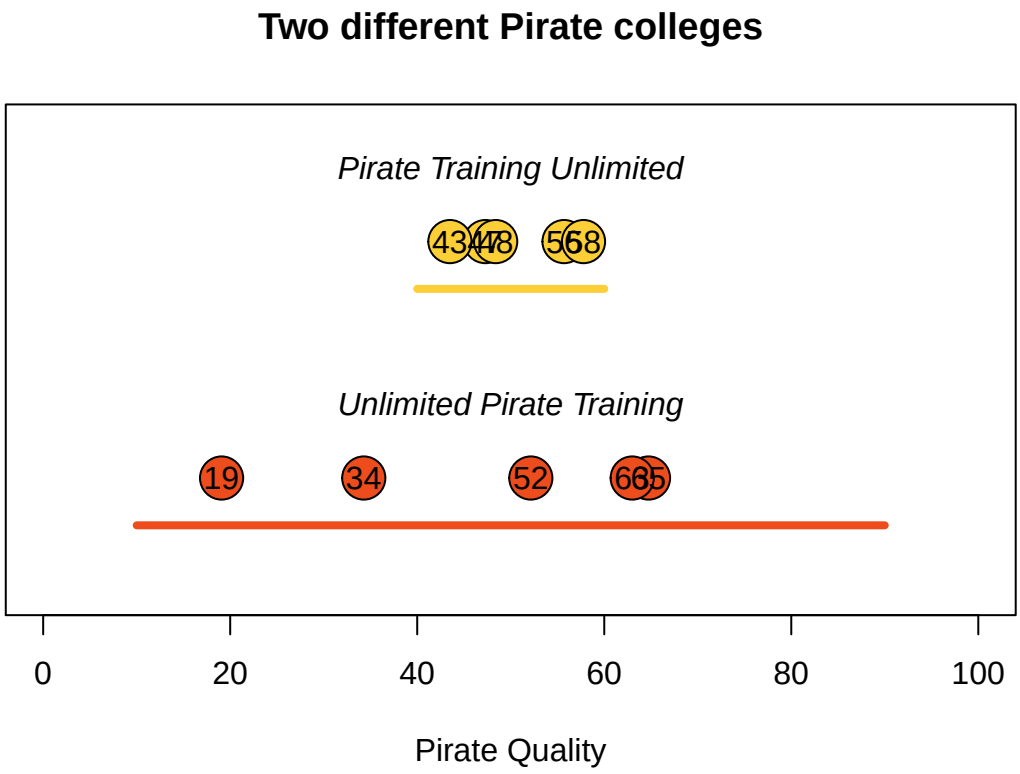


Figure 5.4: Sampling 5 potential pirates from two different pirate colleges. Pirate Training Unlimited (PTU) consistently produces average pirates (with scores between 40 and 60), while Unlimited Pirate Training (UPT), produces a wide range of pirates from 0 to 100.

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

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

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

5.3.2 Normal (Gaussian)

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

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

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

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

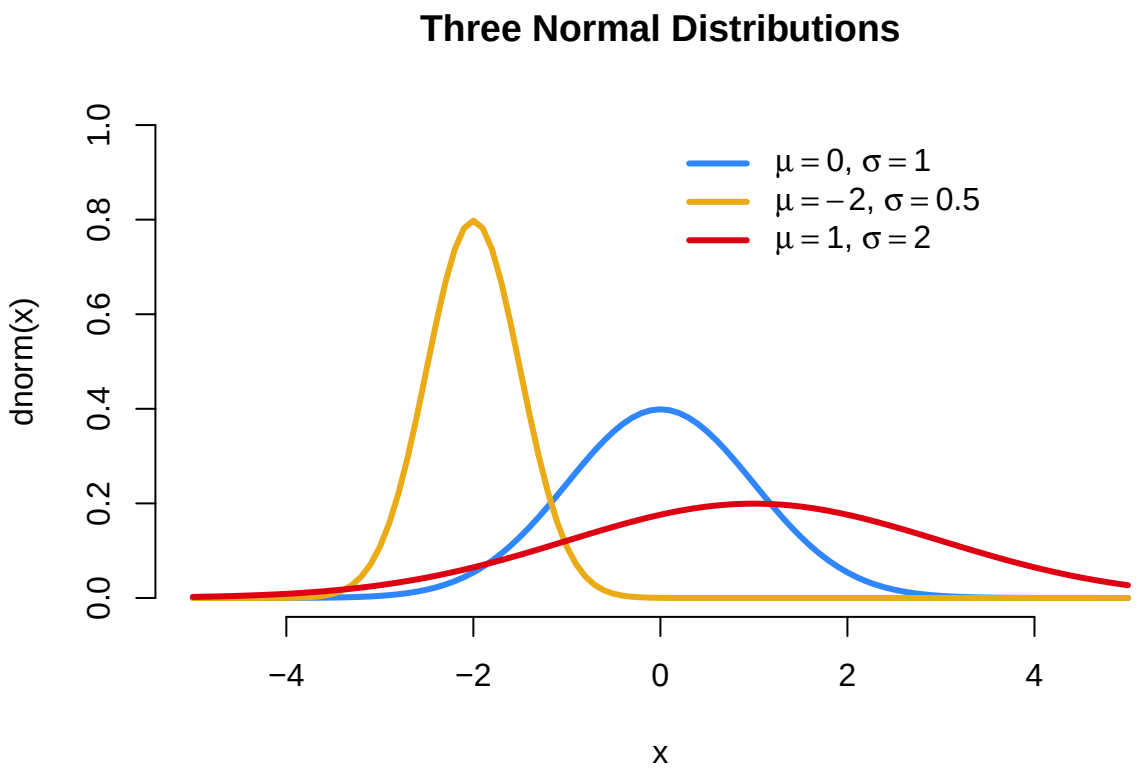


Figure 5.5: Three different normal distributions with different means and standard deviations

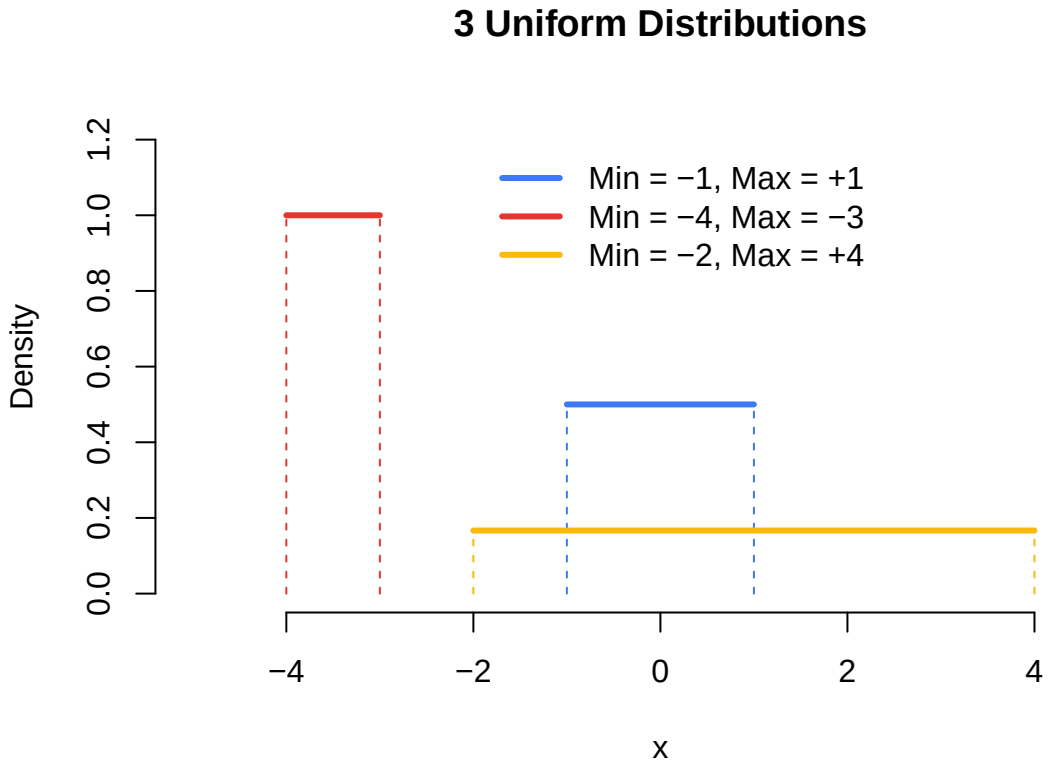


Figure 5.6: The Uniform distribution - known colloquially as the Anthony Davis distribution.

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

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

5.3.4 Notes on random samples

5.3.4.1 Random samples will always change

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

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

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

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

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

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

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

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

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

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

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

5.4 Test your R might!

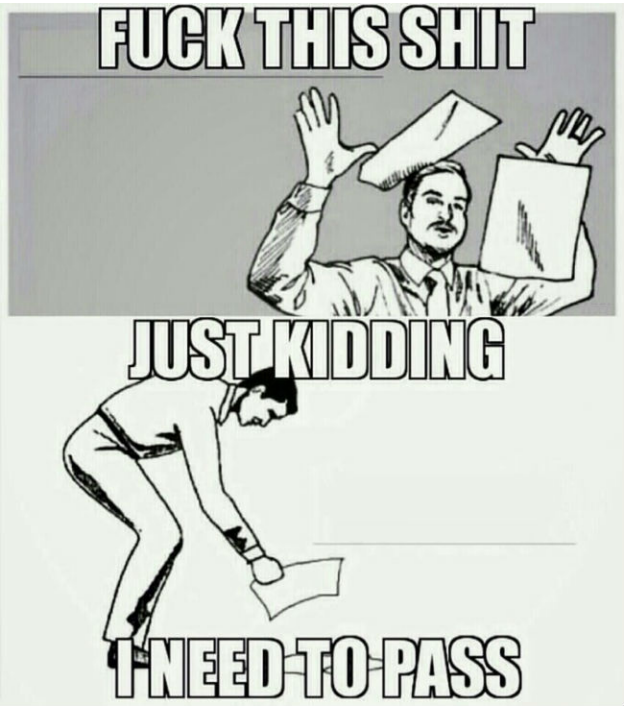
1. Create the vector [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] in three ways: once using `c()`, once using `a:b`, and once using `seq()`.
2. Create the vector [2.1, 4.1, 6.1, 8.1] in two ways, once using `c()` and once using `seq()`
3. Create the vector [0, 5, 10, 15] in 3 ways: using `c()`, `seq()` with a `by`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

Chapter 6

Vector functions

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

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

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

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

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

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

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

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

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

# z-scores
```




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

6.1 Arithmetic operations on vectors

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

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

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

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

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

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

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

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

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

6.1.1 Basic math with multiple vectors

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

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

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

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

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

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

6.1.2 Ex: Pirate Bake Sale

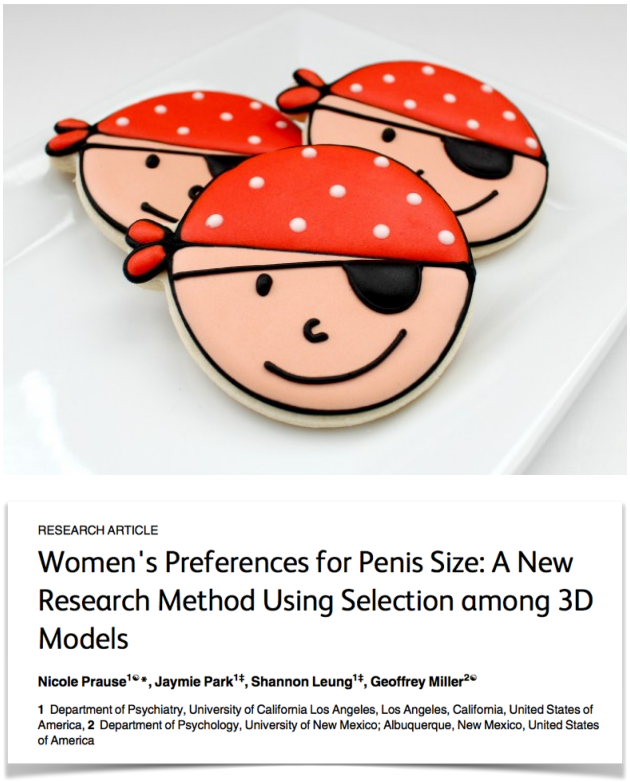
Let's say you had a bake sale on your ship where 5 pirates sold both pies and cookies. You could record the total number of pies and cookies sold in two vectors:

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

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

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

Crazy.



RESEARCH ARTICLE

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

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

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

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

6.2 Summary statistics

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

Table 6.1: Summary statistic functions for continuous data.

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

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

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

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

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

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

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

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

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

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

6.2.2 Additional numeric vector functions

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

Table 6.2: Vector summary functions for continuous data.

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

6.2.3 Sample statistics from random samples

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

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

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

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

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

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

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

6.3 Counting statistics

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

Table 6.3: Counting functions for discrete data.

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

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

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

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

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

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

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

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

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

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

6.4 Missing (NA) values

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

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

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

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

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

6.5 Standardization (z-score)

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

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

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

Table 6.4: Scores from a pirate competition

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

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

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

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

Now let's look at the standardized values:

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

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

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

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

6.5.1 Ex: Evaluating a competition

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

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

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

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

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

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

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

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

Now let's look at the final results

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

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

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

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

Let's look at the result:

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

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

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

6.6 Test your R Might!

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

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

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

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

Chapter 7

Indexing Vectors with [ ]

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

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

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

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

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

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

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

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

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

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

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

7.1 Numerical Indexing

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

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

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

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

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

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

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

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

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

7.2 Logical Indexing

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

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

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

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

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

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

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

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

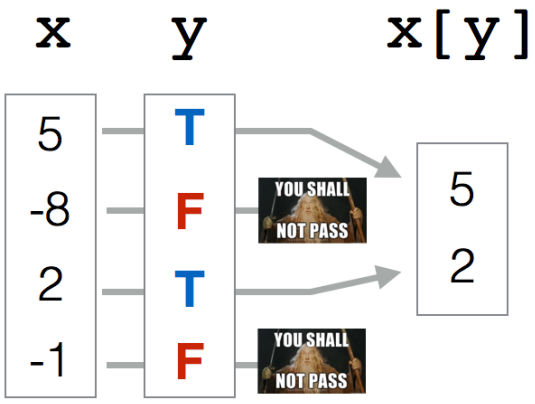


Figure 7.2: FALSE values in a logical vector are like lots of mini-Gandolfs. In this example, I am indexing a vector x with a logical vector y (y for example could be x > 0, so all positive values of x are TRUE and all negative values are FALSE). The result is a vector of length 2, which are the values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

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

Figure 7.3: Logical comparison operators in R

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.2.2 Counts and percentages from logical vectors

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

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

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

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

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

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

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

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

7.2.3 Additional Logical functions

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

Table 7.1: Functions to create and use logical vectors.

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

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

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

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

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

7.3 Changing values of a vector

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

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

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

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

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

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

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

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

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

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

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

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

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

A note on indexing...

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

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

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

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

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

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


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

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

7.3.1 Ex: Fixing invalid responses to a Happiness survey

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

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

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

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

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

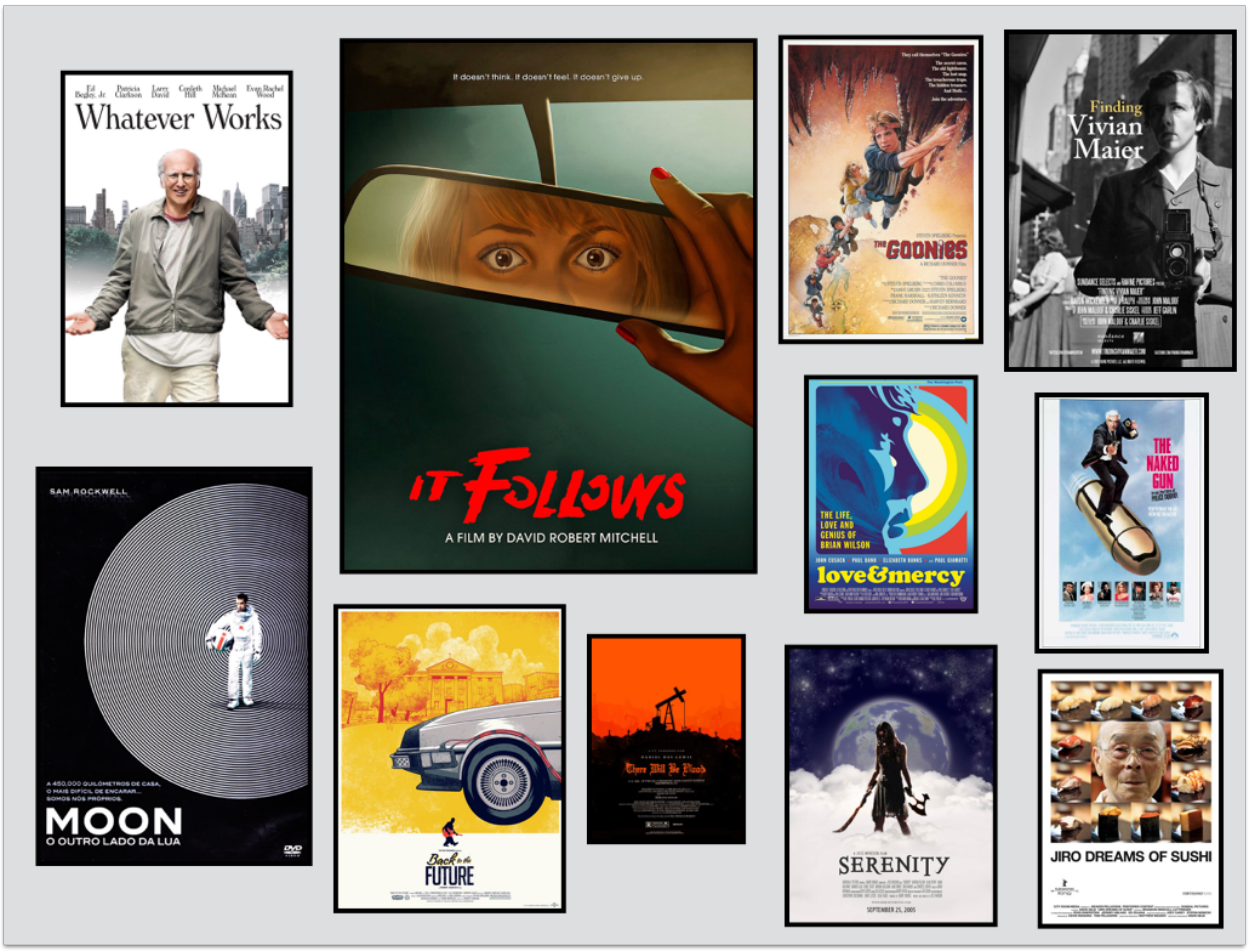
```
# Convert any invalid values in happy to NA
happy[invalid] <- NA
happy
##  [1]  1  4  2 NA  2  3 NA  3  2 NA
```

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

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

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

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



7.4 Test your R Might!: Movie data

Table 7.2 contains data about 10 of my favorite movies.

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

Table 7.2: Some of my favorite movies

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

Chapter 8

Matrices and Dataframes

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

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

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

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

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

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

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

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

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

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

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

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

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

8.1 What are matrices and dataframes?

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

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

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

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

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

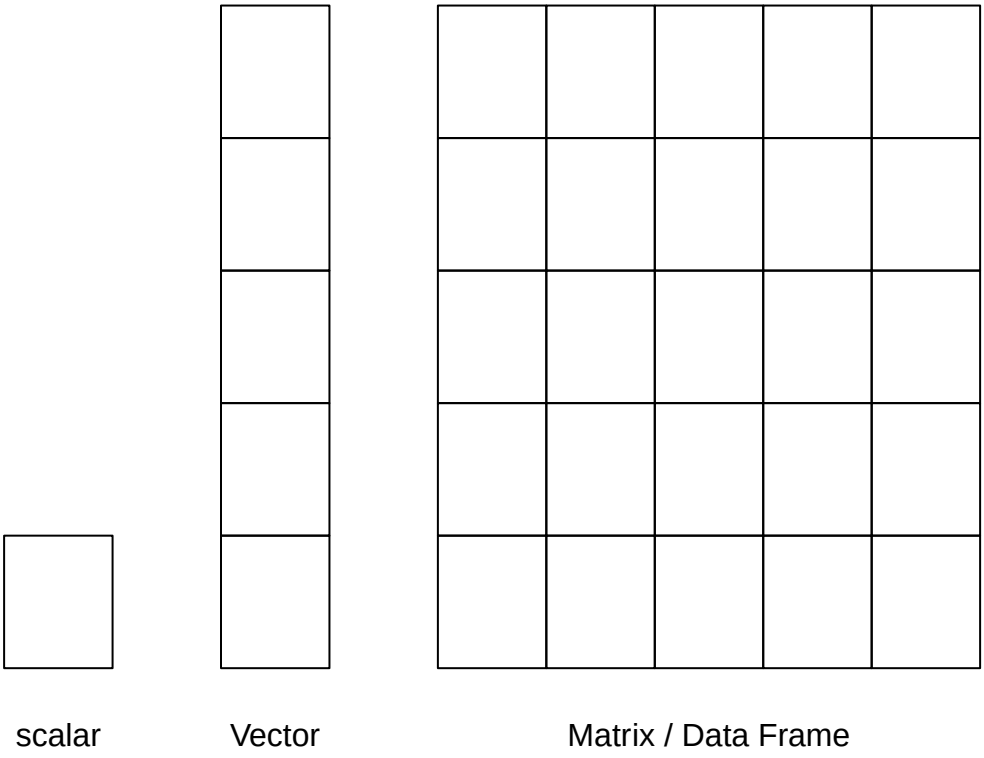


Figure 8.2: scalar, Vector, MATRIX

8.2 Creating matrices and dataframes

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

Table 8.1: Functions to create matrices and dataframes.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.2.3 data.frame()

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

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

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

```
# Create a dataframe of survey data

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

8.2.3.1 `stringsAsFactors = FALSE`

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

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

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

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

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

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

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

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

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

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

8.2.4 Dataframes pre-loaded in R

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

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

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

8.3 Matrix and dataframe functions

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

Table 8.3: Important functions for understanding matrices and dataframes.

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

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

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

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

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

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

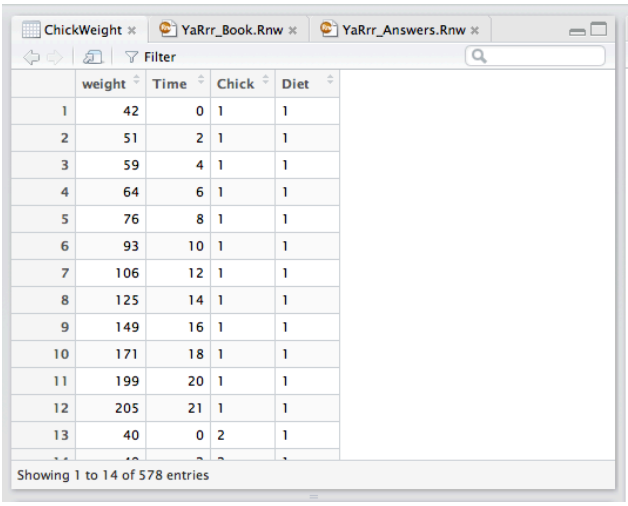
```
# View() opens the entire dataframe in a new window
View(ChickWeight)
```

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

8.3.2 `summary(), str()`

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

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



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

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

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

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

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

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

8.4 Dataframe column names

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

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

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

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

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

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

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

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

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

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

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

8.4.1 Adding new columns

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

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

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

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

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

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

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

Here's the result

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

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

8.4.2 Changing column names

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

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

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

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

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

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

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

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

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

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

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

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

8.5 Slicing dataframes

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

8.5.1 Slicing with `[, ]`

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

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

# Return column 5
df[, 5]

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

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

Table 8.4: First few rows of the ToothGrowth dataframe.

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

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

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

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

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

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

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

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

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

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

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

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

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

8.5.2 Slicing with logical vectors

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

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

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

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

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

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

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

8.5.3 Slicing with `subset()`

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

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

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

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

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

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

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

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

8.6 Combining slicing with functions

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

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

# Step 1: Create a subsettted dataframe called oj

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

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

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

We can also get the same solution using logical indexing:

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

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

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

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

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

8.6.1 `with()`

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

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

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

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

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

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

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

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

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

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

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

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

8.7 Test your R might! Pirates and superheroes

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

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

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

Chapter 9

Importing, saving and managing data

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

9.1 Workspace management functions

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

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

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

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

9.1.1 Why object and file management is so important

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

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

9.2 The working directory

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

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

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

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

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

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

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

9.3 Projects in RStudio

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

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

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

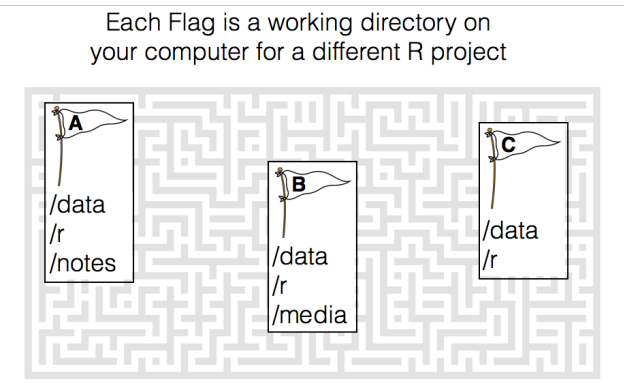


Figure 9.3: A working directory is like a flag on your computer that tells R where to start looking for your files related to a specific project. Each project should have its own folder with organized sub-folders.

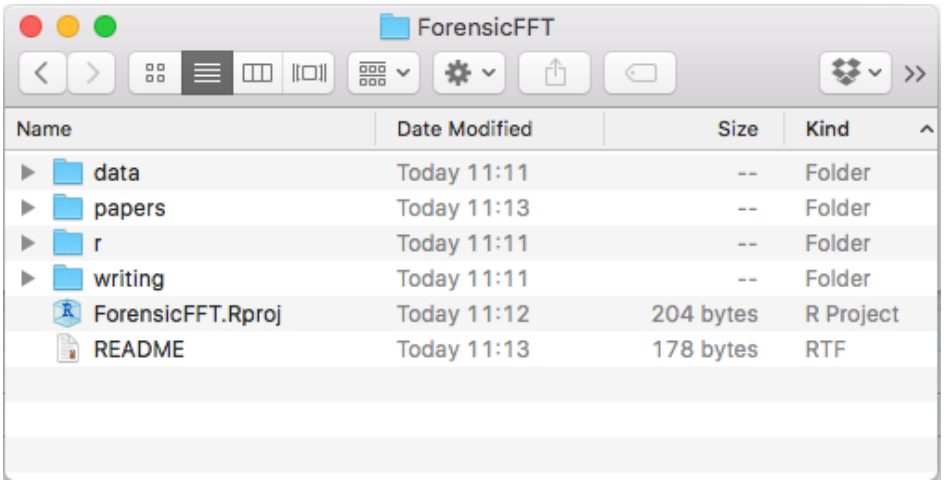


Figure 9.4: Here is the folder structure I use for the working directory in my R project called ForensicFFT. As you can see, it contains an .Rproj file generated by RStudio which sets this folder as the working directory. I also created a folder called r for R code, a folder called data for.txt and .RData files) among others.

by double–clicking on the .Rproj file outside of R), the working directory will automatically be set to the directory that the `.RProj` file is located in.

I recommend creating a new R Project whenever you are starting a new research project. Once you've created a new R project, you should immediately create folders in the directory which will contain your R code, data files, notes, and other material relevant to your project (you can do this outside of R on your computer, or in the Files window of RStudio). For example, you could create a folder called `R` that contains all of your R code, a folder called `data` that contains all your data (etc.). In Figure~9.4 you can see how my working directory looks for a project I am working on called ForensicFFT.

9.4 The workspace

The **workspace** (aka your **working environment**) represents all of the objects and functions you have either defined in the current session, or have loaded from a previous session. When you started RStudio for the first time, the working environment was empty because you hadn't created any new objects or functions. However, as you defined new objects and functions using the assignment operator `<-`, these new

objects were stored in your working environment. When you closed RStudio after defining new objects, you likely got a message asking you "Save workspace image…?"" This is RStudio's way of asking you if you want to save all the objects currently defined in your workspace as an **image file** on your computer.

9.4.1 ls()

If you want to see all the objects defined in your current workspace, use the `ls()` function.

```
# Print all the objects in my workspace
ls()
```

When I run `ls()` I received the following result:

```
## [1] "study1.df" "study2.df" "lm.study1" "lm.study2" "bf.study1"
```

The result above says that I have these 5 objects in my workspace. If I try to refer to an object not listed here, R will return an error. For example, if I try to print `study3.df` (which isn't in my workspace), I will receive the following error:

```
# Try to print study3.df
#  Error because study3.df is NOT in my current workspace
study3.df
```

Error: object 'study3.df' not found

If you receive this error, it's because the object you are referring to is not in your current workspace. 99% of the time, this happens because you mistyped the name of an object.

9.5 .RData files

The best way to store objects from R is with `.RData files`. `.RData` files are specific to R and can store as many objects as you'd like within a single file. Think about that. If you are conducting an analysis with 10 different dataframes and 5 hypothesis tests, you can save **all** of those objects in a single file called `ExperimentResults.RData`.

9.5.1 save()

To save selected objects into one `.RData` file, use the `save()` function. When you run the `save()` function with specific objects as arguments, (like `save(a, b, c, file = "myobjects.RData")` all of those objects will be saved in a single file called `myobjects.RData`

For example, let's create a few objects corresponding to a study.

```
# Create some objects that we'll save later
study1.df <- data.frame(id = 1:5,
                        sex = c("m", "m", "f", "f", "m"),
                        score = c(51, 20, 67, 52, 42))

score.by.sex <- aggregate(score ~ sex,
                          FUN = mean,
                          data = study1.df)

study1.htest <- t.test(score ~ sex,
                       data = study1.df)
```

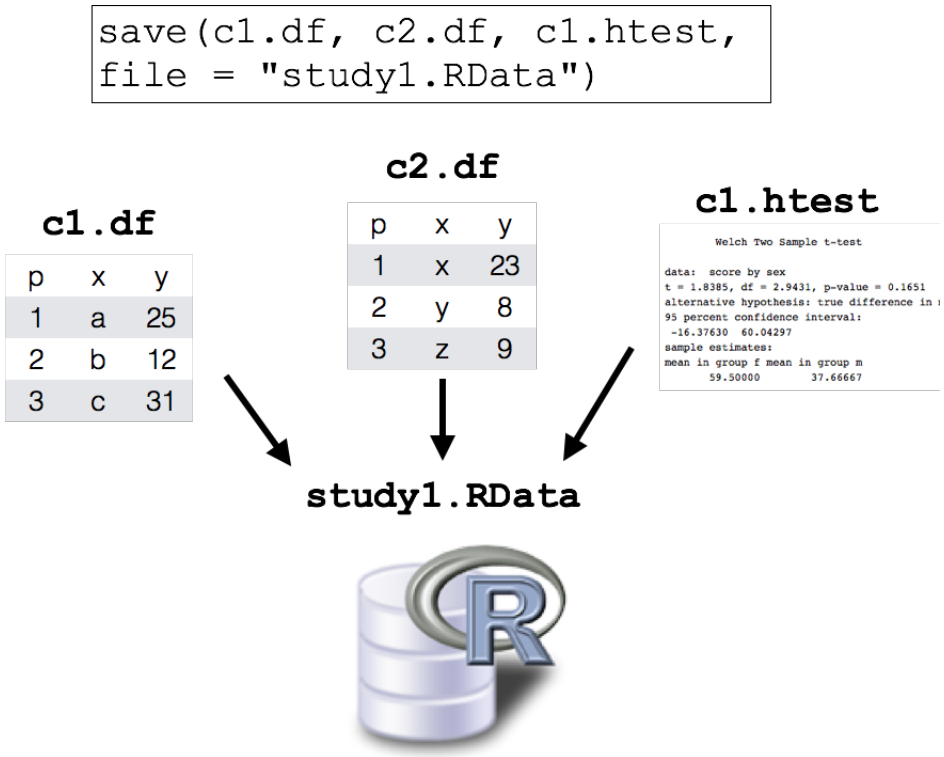


Figure 9.5: Saving multiple objects into a single .RData file.

Now that we've done all of this work, we want to save all three objects in an a file called `study1.RData` in the data folder of my current working directory. To do this, you can run the following

```
# Save two objects as a new .RData file
#   in the data folder of my current working directory
save(study1.df, score.by.sex, study1.htest,
     file = "data/study1.RData")
```

Once you do this, you should see the `study1.RData` file in the data folder of your working directory. This file now contains all of your objects that you can easily access later using the `load()` function (we'll go over this in a second…).

9.5.2 save.image()

If you have many objects that you want to save, then using `save` can be tedious as you'd have to type the name of every object. To save *all* the objects in your workspace as a .RData file, use the `save.image()` function. For example, to save my workspace in the `data` folder located in my working directory, I'd run the following:

```
# Save my workspace to complete_image.RData in th,e
#  data folder of my working directory
save.image(file = "data/projectimage.RData")
```

Now, the `projectimage.RData` file contains *all* objects in your current workspace.

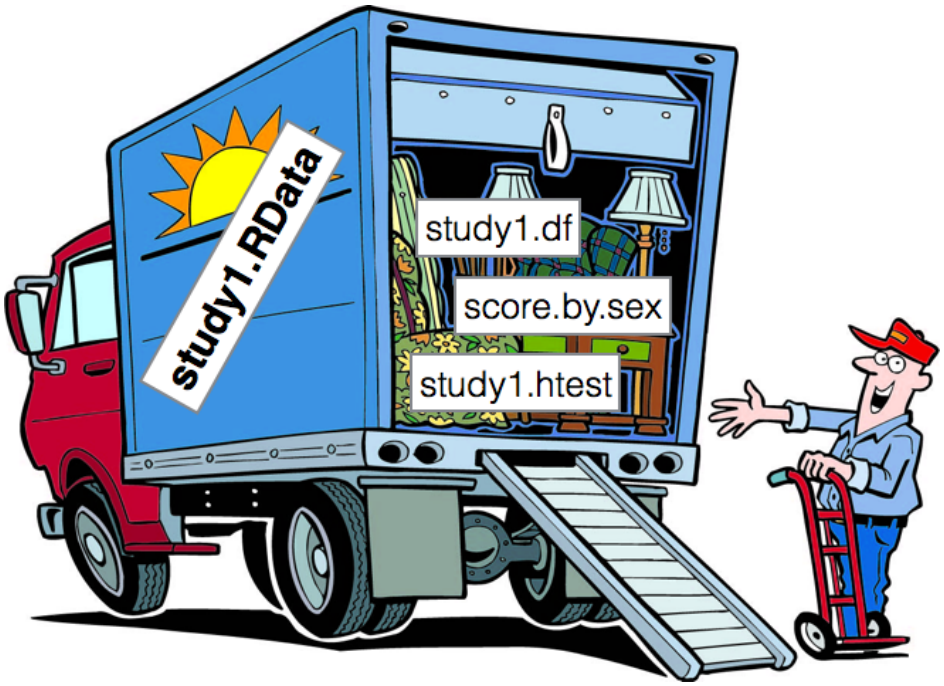


Figure 9.6: Our new study1.RData file is like a van filled with our objects.

9.5.3 load()

To load an `.RData` file, that is, to import all of the objects contained in the `.RData` file into your current workspace, use the `load()` function. For example, to load the three specific objects that I saved earlier (`study1.df`, `score.by.sex`, and `study1.htest`) in `study1.RData`, I'd run the following:

```
# Load objects in study1.RData into my workspace
load(file = "data/study1.RData")
```

To load all of the objects in the workspace that I just saved to the data folder in my working directory in `projectimage.RData`, I'd run the following:

```
# Load objects in projectimage.RData into my workspace
load(file = "data/projectimage.RData")
```

I hope you realize how awesome loading .RData files is. With R, you can store all of your objects, from dataframes to hypothesis tests, in a single `.RData` file. And then load them into any R session at any time using `load()`.

9.5.4 rm()

To remove objects from your workspace, use the `rm()` function. Why would you want to remove objects? At some points in your analyses, you may find that your workspace is filled up with one or more objects that you don't need – either because they're slowing down your computer, or because they're just distracting.

To remove specific objects, enter the objects as arguments to `rm()`. For example, to remove a huge dataframe called `huge.df`, I'd run the following;

```
# Remove huge.df from workspace
rm(huge.df)
```

If you want to remove *all* of the objects in your working directory, enter the argument `list = ls()`

```
# Remove ALL objects from workspace
rm(list = ls())
```

Important!!! Once you remove an object, you **cannot** get it back without running the code that originally generated the object! That is, you can't simply click 'Undo' to get an object back. Thankfully, if your R code is complete and well-documented, you should easily be able to either re-create a lost object (e.g.; the results of a regression analysis), or re-load it from an external file.

9.6 .txt files

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

9.6.1 write.table()

Table 9.2: Arguments for the `write.table()` function that will save an object x (usually a data frame) as a .txt file.

Argument	Description
`x`	The object you want to write to a text file, usually a dataframe
`file`	The document's file path relative to the working directory unless specified otherwise. For example `file = "mydata.txt"` saves the text file directly in the working directory, while `file = "data/mydata.txt"` will save the data in an existing folder called `data` inside the working directory.You can also specify a full file path outside of your working directory (`file = "/Users/CaptainJack/Desktop/OctoberStudy/mydata.txt"`)
`sep`	A string indicating how the columns are separated. For comma separated files, use `sep = ","`, for tab–delimited files, use `sep = "\t"`
`row.names`	A logical value (TRUE or FALSE) indicating whether or not save the rownames in the text file. (`row.names = FALSE` will not include row names)

For example, the following code will save the `pirates` dataframe as a tab–delimited text file called `pirates.txt` in my working directory:

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

write.table(x = pirates,
            file = "pirates.txt",  # Save the file as pirates.txt
            sep = "\t")            # Make the columns tab-delimited
```

If you want to save a file to a location outside of your working directory, just use the entire directory name. When you enter a long path name into the `file` argument of `write.table()`, R will look for that directory outside of your working directory. For example, to save a text file to my Desktop (which is outside of my working directory), I would set `file = "Users/nphillips/Desktop/pirates.txt"`.

```
# Write the pirates dataframe object to a tab-delimited
#  text file called pirates.txt to my desktop

write.table(x = pirates,
            file = "Users/nphillips/Desktop/pirates.txt",  # Save the file as pirates.txt to my desktop
            sep = "\t")                                    # Make the columns tab-delimited
```

9.6.2 read.table()

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

Argument	Description
`file`	The document's file path relative to the working directory unless specified otherwise. For example `file = "mydata.txt"` looks for the text file directly in the working directory, while `file = "data/mydata.txt"` will look for the file in an existing folder called `data` inside the working directory.If the file is outside of your working directory, you can also specify a full file path (`file = "/Users/CaptainJack/Desktop/OctoberStudy/mydata.txt"`)
`header`	A logical value indicating whether the data has a header row – that is, whether the first row of the data represents the column names.
`sep`	A string indicating how the columns are separated. For comma separated files, use `sep = ","`, for tab–delimited files, use `sep = "\t"`
`stringsAsFactors`	A logical value indicating whether or not to convert strings to factors. I **always** set this to FALSE because I *hate*, *hate*, *hate* how R uses factors

The three critical arguments to `read.table()` are `file`, `sep`, `header` and `stringsAsFactors`. The `file` argument is a character value telling R where to find the file. If the file is in a folder in your working directory, just specify the path within your working directory (e.g.; `file = data/newdata.txt`. The `sep` argument tells R how the columns are separated in the file (again, for a comma–separated file, use `sep = ","`}, for a tab–delimited file, use `sep = "\t"`. The `header` argument is a logical value (TRUE or FALSE) telling R whether or not the first row in the data is the name of the data columns. Finally, the `stringsAsFactors` argument is a logical value indicating whether or not to convert strings to factors (I *always* set this to FALSE!)

Let's test this function out by reading in a text file titled `mydata.txt`. Since the text file is located a folder called `data` in my working directory, I'll use the file path `file = "data/mydata.txt"` and since the file is tab–delimited, I'll use the argument `sep = "\t"`:

```
# Read a tab-delimited text file called mydata.txt
#  from the data folder in my working directory into
#  R and store as a new object called mydata

mydata <- read.table(file = 'data/mydata.txt',     # file is in a data folder in my working directory
                     sep = '\t',                   # file is tab--delimited
                     header = TRUE,                # the first row of the data is a header row
                     stringsAsFactors = FALSE)     # do NOT convert strings to factors!!
```

9.6.3 Reading files directly from a web URL

A really neat feature of the `read.table()` function is that you can use it to load text files directly from the web (assuming you are online). To do this, just set the file path to the document's web URL (beginning with `http://`). For example, I have a text file stored at `http://goo.gl/jTNf6P`. You can import and save this tab–delimited text file as a new object called `fromweb` as follows:

```
# Read a text file from the web
fromweb <- read.table(file = 'http://goo.gl/jTNf6P',
                      sep = '\t',
                      header = TRUE)

# Print the result
fromweb
##            message randomdata
## 1 Congratulations!          1
## 2              you          2
## 3             just          3
## 4       downloaded          4
## 5             this          5
## 6            table          6
## 7             from          7
## 8              the          8
## 9             web!          9
```

I think this is pretty darn cool. This means you can save your main data files on Dropbox or a web-server, and always have access to it from any computer by accessing it from its web URL.

Debugging

When you run `read.table()`, you might receive an error like this:

Error in file(file, "rt") : cannot open the connection

In addition: Warning message:

In file(file, "rt") : cannot open file 'asdf': No such file or directory

If you receive this error, it's likely because you either spelled the file name incorrectly, or did not specify the correct directory location in the `file` argument.

9.7 Excel, SPSS, and other data files

A common question I hear is "How can I import an SPSS/Excel/... file into R?". The first answer to this question I always give is "You shouldn't". **S**hitty **P**iece of **S**hitty **S**hit files can contain information like variable descriptions that R doesn't know what to do with, and Excel files often contain something, like missing rows or cells with text instead of numbers, that can completely confuse R.

Rather than trying to import SPSS or Excel files directly in R, I always recommend first exporting/saving the original SPSS or Excel files as text `.txt.` files – both SPSS and Excel have options to do this. Then, once you have exported the data to a `.txt` file, you can read it into R using `read.table()`.

Warning: If you try to export an Excel file to a text file, it is a good idea to clean the file as much as you can first by, for example, deleting unnecessary columns, making sure that all numeric columns have numeric data, making sure the column names are simple (ie., single words without spaces or special

characters). If there is anything 'unclean' in the file, then R may still have problems reading it, even after you export it to a text file.

If you absolutely *have* to read a non-text file into R, check out the package called `foreign` (`install.packages("foreign")`). This package has functions for importing Stata, SAS and SPSS files directly into R. To read Excel files, try the package `xlsx` (`install.packages("xlsx")`). But again, in my experience it's *always* better to convert such files to simple text files first, and then read them into R with `read.table()`.

9.8 Additional tips

1. There are many functions other than `read.table()` for importing data. For example, the functions `read.csv` and `read.delim` are specific for importing comma-separated and tab-separated text files. In practice, these functions do the same thing as `read.table`, but they don't require you to specify a `sep` argument. Personally, I always use `read.table()` because it always works and I don't like trying to remember unnecessary functions.

9.9 Test your R Might!

1. In RStudio, open a new R Project in a new directory by clicking File – New Project. Call the directory `MyRProject`, and then select a directory on your computer for the project. This will be the project's working directory.
2. Outside of RStudio, navigate to the directory you selected in Question 1 and create three new folders – Call them `data`,`R`, and`notes`'.
3. Go back to RStudio and open a new R script. Save the script as `CreatingObjects.R` in the `R` folder you created in Question 2.
4. In the script, create new objects called `a`, `b`, and `c`. You can assign anything to these objects – from vectors to dataframes. If you can't think of any, use these:

```
a <- data.frame("sex" = c("m", "f", "m"),
                "age" = c(19, 43, 25),
                "favorite.movie" = c("Moon", "The Goonies", "Spice World"))
b <- mean(a$age)

c <- table(a$sex)
```

5. Send the code to the Console so the objects are stored in your current workspace. Use the `ls()` function to see that the objects are indeed stored in your workspace.
6. I have a tab–delimited text file called `club` at the following web address: http://nathanieldphillips.com/wp-content/uploads/2015/12/club.txt. Using `read.table()`, load the data as a new object called `club.df` in your workspace.
7. Using `write.table()`, save the dataframe as a tab–delimited text file called `club.txt` to the data folder you created in Question 2. Note: You won't use the text file again for this exercise, but now you have it handy in case you need to share it with someone who doesn't use R.
8. Save the three objects `a`, `b`, `c`, and `club.df` to an .RData file called "myobjects.RData" in your data folder using `save()`.
9. Clear your workspace using the `rm(list = ls())` function. Then, run the `ls()` function to make sure the objects are gone.

10. Open a new R script called `AnalyzingObjects.R` and save the script to the `R` folder you created in Question 2.
11. Now, in your `AnalyzingObjects.R` script, load the objects back into your workspace from the `myobjects.RData` file using the `load()` function. Again, run the `ls()` function to make sure all the objects are back in your workspace.
12. Add some R code to your `AnalyzingObjects.R` script. Calculate some means and percentages. Now save your `AnalyzingObjects.R` script, and then save all the objects in your workspace to `myobjects.RData`.
13. Congratulations! You are now a well-organized R Pirate! Quit RStudio and go outside for some relaxing pillaging.

Chapter 10

Advanced dataframe manipulation

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

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

# Demographic data
demographics <- data.frame(
  id = 1:5,
  sex = c("f", "m", "f", "f", "m"),
  age = c(25, 22, 24, 19, 23))

# Combine exam and demographics
combined <- merge(x = exam,
                  y = demographics,
                  by = "id")

# Mean q1 score for each sex
aggregate(formula = q1 ~ sex,
          data = combined,
          FUN = mean)
##   sex  q1
## 1   f 2.0
```

Figure 10.1: Make your dataframes dance for you

```
## 2   m 3.5

# Median q3 score for each sex, but only for those
#   older than 20
aggregate(formula = q3 ~ sex,
          data = combined,
          subset = age > 20,
          FUN = mean)
##   sex  q3
## 1   f 3.5
## 2   m 5.5

# Many summary statistics by sex using dplyr!
library(dplyr)
combined %>% group_by(sex) %>%
  summarise(
    q1.mean = mean(q1),
    q2.mean = mean(q2),
    q3.mean = mean(q3),
    age.mean = mean(age),
    N = n())
## # A tibble: 2 x 6
##      sex q1.mean q2.mean q3.mean age.mean     N
##   <fctr>   <dbl>   <dbl>   <dbl>    <dbl> <int>
## 1      f     2.0     8.3     4.3       23     3
## 2      m     3.5     8.5     5.5       22     2
```

In Chapter 6, you learned how to calculate statistics on subsets of data using indexing. However, you may have noticed that indexing is not very intuitive and not terribly efficient. If you want to calculate statistics for many different subsets of data (e.g.; mean birth rate for every country), you'd have to write a new indexing command for each subset, which could take forever. Thankfully, R has some great built-in functions like `aggregate()` that allow you to easily apply functions (like `mean()`) to a dependent variable (like birth rate) for each level of one or more independent variables (like a country) with just a few lines of code.

10.1 `order()`: Sorting data

To sort the rows of a dataframe according to column values, use the `order()` function. The `order()` function takes one or more vectors as arguments, and returns an integer vector indicating the order of the vectors. You can use the output of `order()` to index a dataframe, and thus change its order.

Let's re-order the `pirates` data by height from the shortest to the tallest:

```
# Sort the pirates dataframe by height
pirates <- pirates[order(pirates$height),]

# Look at the first few rows and columns of the result
pirates[1:5, 1:4]
##      id    sex age height
## 39   39 female  25    130
## 854 854 female  25    130
## 30   30 female  26    135
## 223 223 female  28    135
```

```
## 351 351 female  36    137
```

By default, the `order()` function will sort values in ascending (increasing) order. If you want to order the values in descending (decreasing) order, just add the argument `decreasing = TRUE` to the `order()` function:

```
# Sort the pirates dataframe by height in decreasing order
pirates <- pirates[order(pirates$height, decreasing = TRUE),]

# Look at the first few rows and columns of the result
pirates[1:5, 1:4]
##      id  sex age height
## 2     2 male  31    209
## 793 793 male  25    209
## 430 430 male  26    201
## 292 292 male  29    201
## 895 895 male  27    201
```

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

```
# Sort the pirates dataframe by sex and then height
pirates <- pirates[order(pirates$sex, pirates$height),]
```

By default, the `order()` function will sort values in ascending (increasing) order. If you want to order the values in descending (decreasing) order, just add the argument `decreasing = TRUE` to the `order()` function:

```
# Sort the pirates dataframe by height in decreasing order
pirates <- pirates[order(pirates$height, decreasing = TRUE),]
```

10.2 merge(): Combining data

Argument	Description
`x, y`	Two dataframes to be merged
`by`	A string vector of 1 or more columns to match the data by. For example, `by = "id"` will combine columns that have matching values in a column called `"id"`. `by = c("last.name", "first.name")` will combine columns that have matching values in both `"last.name"` and `"first.name"`
`all`	A logical value indicating whether or not to include rows with non-matching values of `by`.

One of the most common data management tasks is **merging** (aka combining) two data sets together. For example, imagine you conduct a study where 5 participants are given a score from 1 to 5 on a risk assessment task. We can represent these data in a dataframe called `risk.survey`:

```
# Results from a risk survey
risk.survey <- data.frame(
  "participant" = c(1, 2, 3, 4, 5),
  "risk.score" = c(3, 4, 5, 3, 1))
```

Now, imagine that in a second study, you have participants complete a survey about their level of happiness

Table 10.2: Results from a survey on risk.

participant	risk.score
1	3
2	4
3	5
4	3
5	1

(on a scale of 0 to 100). We can represent these data in a new dataframe called `happiness.survey`:

```
happiness.survey <- data.frame(
  "participant" = c(4, 2, 5, 1, 3),
  "happiness.score" = c(20, 40, 50, 90, 53))
```

Now, we'd like to combine these data into one data frame so that the two survey scores for each participant are contained in one object. To do this, use `merge()`.

When you merge two dataframes, the result is a new dataframe that contains data from both dataframes. The key argument in `merge()` is `by`. The `by` argument specifies how rows should be matched during the merge. Usually, this will be something like an name, id number, or some other unique identifier.

Let's combine our risk and happiness survey using `merge()`. Because we want to match rows by the `participant.id` column, we'll specify `by = "participant.id"`. Additionally, because we want to include rows with potentially non-matching values, we'll include `all = TRUE`

```
# Combine the risk and happiness surveys by matching participant.id
combined.survey <- merge(x = risk.survey,
                         y = happiness.survey,
                         by = "participant",
                         all = TRUE)

# Print the result
combined.survey
##   participant risk.score happiness.score
## 1           1          3              90
## 2           2          4              40
## 3           3          5              53
## 4           4          3              20
## 5           5          1              50
```

For the rest of the chapter, we'll cover data aggregation functions. These functions allow you to quickly and easily calculate aggregated summary statistics over groups of data in a data frame. For example, you can use them to answer questions such as "What was the mean crew age for each ship?", or "What percentage of participants completed an attention check for each study condition?" We'll start by going over the `aggregate()` function.

10.3 `aggregate()`: Grouped aggregation

Argument	Description
`formula`	A formula in the form `y ~ x1 + x2 + ...` where y is the dependent variable, and x1, x2… are the independent variables. For example, `salary ~ sex + age` will aggregate a `salary` column at every unique combination of `sex` and `age`
`FUN`	A function that you want to apply to y at every level of the independent variables. E.g.; `mean`, or `max`.
`data`	The dataframe containing the variables in `formula`
`subset`	A subset of data to analyze. For example, `subset(sex == "f" & age > 20)` would restrict the analysis to females older than 20. You can ignore this argument to use all data.

The first aggregation function we'll cover is `aggregate()`. Aggregate allows you to easily answer questions in the form: "What is the value of the function `FUN` applied to a dependent variable `dv` at each level of one (or more) independent variable(s) `iv`?

```
# General structure of aggregate()
aggregate(formula = dv ~ iv, # dv is the data, iv is the group
          FUN = fun, # The function you want to apply
          data = df) # The dataframe object containing dv and iv
```

Let's give `aggregate()` a whirl. No…not a whirl…we'll give it a spin. Definitely a spin. We'll use `aggregate()` on the `ChickWeight` dataset to answer the question "What is the mean weight for each diet?"

If we wanted to answer this question using basic R functions, we'd have to write a separate command for each supplement like this:

```
# The WRONG way to do grouped aggregation.
#  We should be using aggregate() instead!
mean(ChickWeight$weight[ChickWeight$Diet == 1])
## [1] 103
mean(ChickWeight$weight[ChickWeight$Diet == 2])
## [1] 123
mean(ChickWeight$weight[ChickWeight$Diet == 3])
## [1] 143
mean(ChickWeight$weight[ChickWeight$Diet == 4])
## [1] 135
```

If you are ever writing code like this, there is almost always a simpler way to do it. Let's replace this code with a much more elegant solution using `aggregate()`.For this question, we'll set the value of the dependent variable Y to `weight`, x1 to `Diet`, and FUN to `mean`

```
# Calculate the mean weight for each value of Diet
aggregate(formula = weight ~ Diet,  # DV is weight, IV is Diet
          FUN = mean,               # Calculate the mean of each group
          data = ChickWeight)       # dataframe is ChickWeight
##   Diet weight
## 1    1    103
## 2    2    123
## 3    3    143
## 4    4    135
```

As you can see, the `aggregate()` function has returned a dataframe with a column for the independent variable `Diet`, and a column for the results of the function `mean` applied to each level of the independent variable. The result of this function is the same thing we'd got from manually indexing each level of `Diet` individually – but of course, this code is much simpler and more elegant!

You can also include a `subset` argument within an `aggregate()` function to apply the function to subsets of the original data. For example, if I wanted to calculate the mean chicken weights for each diet, but only when the chicks are less than 10 weeks old, I would do the following:

```
# Calculate the mean weight for each value of Diet,
#  But only when chicks are less than 10 weeks old

aggregate(formula = weight ~ Diet,  # DV is weight, IV is Diet
          FUN = mean,               # Calculate the mean of each group
          subset = Time < 10,       # Only when Chicks are less than 10 weeks old
          data = ChickWeight)       # dataframe is ChickWeight
##   Diet weight
## 1    1     58
## 2    2     63
## 3    3     66
## 4    4     69
```

You can also include multiple independent variables in the formula argument to `aggregate()`. For example, let's use `aggregate()` to now get the mean weight of the chicks for all combinations of both `Diet` and `Time`, but now only for weeks 0, 2, and 4:

```
# Calculate the mean weight for each value of Diet and Time,
#  But only when chicks are 0, 2 or 4 weeks okd

aggregate(formula = weight ~ Diet + Time,  # DV is weight, IVs are Diet and Time
          FUN = mean,                      # Calculate the mean of each group
          subset = Time %in% c(0, 2, 4),   # Only when Chicks are 0, 2, and 4 weeks old
          data = ChickWeight)              # dataframe is ChickWeight
##    Diet Time weight
## 1     1    0     41
## 2     2    0     41
## 3     3    0     41
## 4     4    0     41
## 5     1    2     47
## 6     2    2     49
## 7     3    2     50
## 8     4    2     52
## 9     1    4     56
## 10    2    4     60
## 11    3    4     62
## 12    4    4     64
```

10.4 dplyr

The `dplyr` package is a relatively new R package that allows you to do all kinds of analyses quickly and easily. It is especially useful for creating tables of summary statistics across specific groups of data. In this section, we'll go over a very brief overview of how you can use dplyr to easily do grouped aggregation. Just to be clear - you can use dplyr to do everything the `aggregate()` function does and much more! However, this will be a very brief overview and I strongly recommend you look at the help menu for dplyr for additional descriptions and examples.

To use the dplyr package, you first need to install it with `install.packages()` and load it:

```
install.packages("dplyr")     # Install dplyr (only necessary once)
library("dplyr")              # Load dplyr
```

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

To aggregate data with `dplyr`, your code will look something like the following code. In this example, assume that the dataframe you want to summarize is called `my.df`, the variable you want to group the data by independent variables `iv1, iv2`, and the columns you want to aggregate are called `col.a`, `col.b` and `col.c`

```
# Template for using dplyr
my.df %>%                  # Specify original dataframe
  filter(iv3 > 30) %>%     # Filter condition
  group_by(iv1, iv2) %>%   # Grouping variable(s)
  summarise(
    a = mean(col.a),       # calculate mean of column col.a in my.df
    b = sd(col.b),         # calculate sd of column col.b in my.df
    c = max(col.c))        # calculate max on column col.c in my.df, ...
```

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

Let's start with an example: Let's create a dataframe of aggregated data from the `pirates` dataset. I'll filter the data to only include pirates who wear a headband. I'll group the data according to the columns `sex` and `college`. I'll then create several columns of different summary statistic of some data across each grouping. To create this aggregated data frame, I will use the new function `group_by` and the verb `summarise`. I will assign the result to a new dataframe called `pirates.agg`:

```
pirates.agg <- pirates %>%                  # Start with the pirates dataframe
               filter(headband == "yes") %>% # Only pirates that wear hb
               group_by(sex, college) %>%    # Group by these variables
               summarise(
                        age.mean = mean(age),      # Define first summary...
                        tat.med = median(tattoos), # you get the idea...
                        n = n()                    # How many are in each group?
               ) # End

# Print the result
pirates.agg
## # A tibble: 6 x 5
## # Groups:   sex [?]
##      sex college age.mean tat.med     n
##    <chr>   <chr>    <dbl>   <dbl> <int>
## 1 female    CCCC       26      10   206
## 2 female   JSSFP       34      10   203
## 3   male    CCCC       23      10   358
## 4   male   JSSFP       32      10    85
## 5  other    CCCC       25      10    24
## 6  other   JSSFP       32      12    11
```

As you can see from the output on the right, our final object `pirates.agg` is the aggregated dataframe we want which aggregates all the columns we wanted for each combination of `sex` and `college` One key new function here is `n()`. This function is specific to dplyr and returns a frequency of values in a summary command.

Let's do a more complex example where we combine multiple verbs into one chunk of code. We'll aggregate data from the movies dataframe.

```
movies %>% # From the movies dataframe...
    filter(genre != "Horror" & time > 50) %>% # Select only these rows
    group_by(rating, sequel) %>% # Group by rating and sequel
    summarise( #
      frequency = n(), # How many movies in each group?
      budget.mean = mean(budget, na.rm = T),  # Mean budget?
      revenue.mean = mean(revenue.all), # Mean revenue?
      billion.p = mean(revenue.all > 1000)) # Percent of movies with revenue > 1000?
## # A tibble: 14 x 6
## # Groups:   rating [?]
##       rating sequel frequency budget.mean revenue.mean billion.p
##        <chr>  <int>     <int>       <dbl>        <dbl>     <dbl>
##  1         G      0        59       41.23          234    0.0000
##  2         G      1        12       92.92          357    0.0833
##  3     NC-17      0         2        3.75           18    0.0000
##  4 Not Rated      0        84        1.74           56    0.0000
##  5 Not Rated      1        12        0.67           66    0.0000
##  6        PG      0       312       51.78          191    0.0096
##  7        PG      1        62       77.21          372    0.0161
##  8     PG-13      0       645       52.09          168    0.0062
##  9     PG-13      1       120      124.16          524    0.1167
## 10         R      0       623       31.38          109    0.0000
## 11         R      1        42       58.25          226    0.0000
## 12      <NA>      0        86        1.65           34    0.0000
## 13      <NA>      1        15        5.51           48    0.0000
## 14      <NA>     NA        11        0.00           34    0.0000
```

As you can see, our result is a dataframe with 14 rows and 6 columns. The data are summarized from the movie dataframe, only include values where the genre is *not* Horror and the movie length is longer than 50 minutes, is grouped by rating and sequel, and shows several summary statistics.

10.4.1 Additional dplyr help

We've only scratched the surface of what you can do with `dplyr`. In fact, you can perform almost all of your R tasks, from loading, to managing, to saving data, in the `dplyr` framework. For more tips on using dplyr, check out the dplyr vignette at https://cran.r-project.org/web/packages/dplyr/vignettes/introduction.html. Or open it in RStudio by running the following command:

```
# Open the dplyr introduction in R
vignette("introduction", package = "dplyr")
```

There is also a very nice YouTube video covering `dplyr` at https://goo.gl/UY2AE1. Finally, consider also reading R for Data Science written by Garrett Grolemund and Hadley Wickham, which teaches R from the ground-up using the dplyr framework.

10.5 Additional aggregation functions

There are many, many other aggregation functions that I haven't covered in this chapter – mainly because I rarely use them. In fact, that's a good reminder of a peculiarity about R, there are many methods to

Table 10.4: Scores from an exam.

q1	q2	q3	q4	q5
1	1	1	1	1
0	0	0	1	0
0	1	1	1	0
0	1	0	1	1
0	0	0	1	1

achieve the same result, and your choice of which method to use will often come down to which method you just like the most.

10.5.1 `rowMeans()`, `colMeans()`

To easily calculate means (or sums) across all rows or columns in a matrix or dataframe, use `rowMeans()`, `colMeans()`, `rowSums()` or `colSums()`.

For example, imagine we have the following data frame representing scores from a quiz with 5 questions, where each row represents a student, and each column represents a question. Each value can be either 1 (correct) or 0 (incorrect)

```
# Some exam scores
exam <- data.frame("q1" = c(1, 0, 0, 0, 0),
                   "q2" = c(1, 0, 1, 1, 0),
                   "q3" = c(1, 0, 1, 0, 0),
                   "q4" = c(1, 1, 1, 1, 1),
                   "q5" = c(1, 0, 0, 1, 1))
```

Let's use `rowMeans()` to get the average scores for each student:

```
# What percent did each student get correct?
rowMeans(exam)
## [1] 1.0 0.2 0.6 0.6 0.4
```

Now let's use `colMeans()` to get the average scores for each *question*:

```
# What percent of students got each question correct?
colMeans(exam)
##  q1  q2  q3  q4  q5
## 0.2 0.6 0.4 1.0 0.6
```

Warning `rowMeans()` and `colMeans()` only work on numeric columns. If you try to apply them to non-numeric data, you'll receive an error.

10.5.2 `apply` family

There is an entire class of `apply` functions in R that apply functions to groups of data. For example, `tapply()`, `sapply()` and `lapply()` each work very similarly to `aggregate()`. For example, you can calculate the average length of movies by genre with `tapply()` as follows.

```
with(movies, tapply(X = time,        # DV is time
                    INDEX = genre,   # IV is genre
                    FUN = mean,      # function is mean
                    na.rm = TRUE))   # Ignore missing
```

```
##              Action           Adventure        Black Comedy
##                 113                 106                 113
##              Comedy Concert/Performance         Documentary
##                  99                  78                  69
##               Drama              Horror     Multiple Genres
##                 116                  99                 114
##             Musical             Reality     Romantic Comedy
##                 113                  44                 107
##   Thriller/Suspense             Western
##                 112                 121
```

`tapply()`, `sapply()`, and `lapply()` all work very similarly, their main difference is in the structure of their output. For example, `lapply()` returns a **list** (we'll cover lists in a future chapter).

10.6 Test your R might!: Mmmmm...caffeine

You're in charge of analyzing the results of an experiment testing the effects of different forms of caffeine on a measure of performance. In the experiment, 100 participants were given either Green tea or coffee, in doses of either 1 or 5 servings. They then performed a cognitive test where higher scores indicate better performance.

The data are stored in a tab–delimited dataframe at the following link:
https://raw.githubusercontent.com/ndphillips/ThePiratesGuideToR/master/data/caffeinestudy.txt

1. Load the dataset from https://raw.githubusercontent.com/ndphillips/ThePiratesGuideToR/master/data/caffeinestudy.txt as a new dataframe called `caffeine`.
2. Calculate the mean age for each gender
3. Calculate the mean age for each drink
4. Calculate the mean age for each combined level of both gender and drink
5. Calculate the median score for each age

6. For men only, calculate the maximum score for each age
7. Create a dataframe showing, for each level of drink, the mean, median, maximum, and standard deviation of scores.
8. Only for females above the age of 20, create a table showing, for each combined level of drink and cups, the mean, median, maximum, and standard deviation of scores. Also include a column showing how many people were in each group.