

The Worst Stats Text eveR

Dan Stich

Contents

The Worst Stats Text ever	5
Preface	7
About the author	9
1 Introduction to programming in R	11
1.1 What is R?	11
1.2 Why should I use R?	12
1.3 Where do I start?	12
1.4 Programming conventions	13
1.5 Next steps	17
2 Data structures	19
2.1 Vectors	19
2.2 Vector operations	24
2.3 Matrices	27
2.4 Dataframes	28
2.5 Lists	35
2.6 Next steps	37
3 Working with data	39
3.1 Data read	39
3.2 Quick data summaries	41

3.3	Subsetting and selecting data	42
3.4	Better data summaries	45
3.5	Creating new variables	47
3.6	Data simulation	49
3.7	Next steps	52
4	Plotting and graphics	53
4.1	Plots matter as much as stats	54
4.2	Plotting with base R	54
4.3	Plotting with <code>ggplot2</code>	66
4.4	Next steps	79
5	Sampling distributions in R	81
5.1	What are sampling distributions?	82
5.2	Probability distributions in R	82
5.3	Exponential family	83
5.4	Continuous distributions	83
5.5	Discrete distributions	91
5.6	Sample statistics	105
5.7	Next steps	111
6	Inferential statistics	113
6.1	One-sample tests	114
6.2	Two-sample tests	121
6.3	Frequency analysis	127
6.4	Next steps	130

The Worst Stats Text eveR

Dan Stich, PhD

Biology Department and Biological Field Station, SUNY Oneonta

Unskillful representation of an alcohol molecule where -OH is the functional group and R is the radical group, or “rest of the molecule”, much like it is to modern statistics. This is funny, because R is a “functional” programming language that will drive you to drink (or perhaps undertake some other, healthier stress-reducing activity). Don’t worry, I’ll explain all of the jokes, and **most** of the code as we go, because this

is **The Worst Stats Text eveR**.

Preface

This book is a compilation of teaching content and lab activities that I have amassed like a digital hoarder during my time teaching BIOL 217 (Quantitative Biology) at SUNY Oneonta. The book started as a collection of R scripts that I eventually converted into web-pages under the former BIOL 217 course website using rmarkdown, and now finally into an e-document (thanks to bookdown!) that is without doubt The Worst Stats Text everR.

The purpose of this book is to provide a tutorial survey of commonly used statistical tools in R for undergraduate students interested in biology. On any given week, our focus will be to demonstrate one or more techniques in R and show how they might be applied to real-world data, warts and all. My hope is that students take away 1) why we use these tools, 2) how to use them (and how not to!), and 3) how we show what it means. Along the way, we'll incorporate data management and exploration, statistical assumptions, and plotting.

To that end, certain ideas and language within this book are simplified for the target audience - apologies in advance if simplicity or informality jeopardize accuracy in any way. I am happy to receive constructive advice through the GitHub repository for this project located here.

This text and the course assume minimal starting knowledge of statistics or computer programming. We build on both during each chapter, and from one chapter to the next. Throughout the book, we will demonstrate statistical and biological concepts using real and simulated data sets from a variety of sub-disciplines within the biological sciences. My own academic interests are in quantitative aspects of applied ecology and fisheries management. Therefore, many of our examples have a fishy flavor, but I try to incorporate examples from other realms of biology.

The purpose of this book is not to serve as a stand-alone, citable reference document or a comprehensive guide to R even for students enrolled in my own class. It is The Worst Stats Text everR! Why would you cite a book with that name? The code and generally citation-free ranting contained herein are, however, extensively supplemented by targeted readings on each topic from the

primary literature, published text supplements and discussions. The reader is strongly encouraged to seek out other learning resources appropriate to their comfort level (see Additional Resources on the course website).

About the author

Dr. Dan Stich is Assistant Professor of Biology at the State University of New York College at Oneonta. He teaches undergraduate and graduate courses in organismal biology, ichthyology, ecology, experimental design, lake management, and quantitative biology. He also teaches R workshops for various professional societies to which he belongs. His research focuses on the development and application of quantitative models to answer theoretical and applied questions related to fisheries and aquatic resource management and decision making. You can learn more about his teaching and research through his website.

Dan is not a programmer or a statistician. He is a fish scientist who went rogue with code and stumbled into population modeling as a graduate student. At some point it became as much a hobby as a work endeavor. He is an active user of R and Rstudio and delites in seeing others get hooked on it, too. He maintains and contributes to multiple R packages as part of his research. You can find some of these in his GitHub repositories, where he spends much time talking to himself in his own online issue trackers.

Chapter 1

Introduction to programming in R

Title image. Read about it there.

Welcome to programming in R! This module will serve as a tutorial to help you get acquainted with the R programming environment, and will get you started with some basic tools and information that will help you along your way.

We will use the Rstudio IDE to work with R in this class. It is important to note here that R is the program doing all of the thinking when we write and run code, and RStudio is a software tool that makes it a little easier to work with R - so we're going to need them both (plus a few other tools we'll check out along the way).

1.1 What is R?

Go Google it. This is The Worst Stats Text ever.

Okay, okay. Briefly, R is a statistical programming language. That language is made up of functions and various objects (R is functional and object-oriented). Objects are things that we do stuff to, or that we create by doing stuff. Functions are the things that do stuff to objects or create objects by doing things. A lot of functions and objects are included in the **base** software distribution (this is the one you just downloaded). Other collections of functions and objects are available through “packages”. You could think of these packages like web-extensions, add-ins for Microsoft programs, or mods for Minecraft. These packages may be written in R or built on a variety of other programming languages you may have heard of like C, C++, java, Python, etc. You can see a

YouTube demo of installing packages in RStudio [here](#). We will talk more about this later.

Because R is open-source anybody can write packages (even me). Therefore, there are lots of packages out there and many of them have functions that do the same thing but have slightly different names or behaviors. This framework, and an avid user-community has propelled the capabilities of R and RStudio during recent years, and now R can do everything from basic arithmetic to spatial time-series analysis to searching Amazon. This means that learning R is also a lot like learning the English language because there are about 10 ways to do everything and many of those are based on other programming languages.

1.2 Why should I use R?

For now: because this whole class revolves around your using R. If you don't, you'll fail or look silly at a job interview. I started using R because I needed it to finish my master's thesis. I'd like to think some people start using R "just because" they want to, but usually those people just say they want to start it.

Students are in a unique position to be able to do the things they want to do because they have to do them (somebody write that down). Most of us should probably make it more of a priority.

On that note, hopefully the "why" becomes obvious to you during our time together even if you don't want to be a data scientist or a modeler. If you only ever use R to do t-tests or make descriptive plots it is worth learning. The ability to re-use the same code for a later analysis alone can save you hours. You never lose what you write (and back up!). So, the more and the longer you write R code, the more time you will have to do other things in life that you care more about (as if). If R *is* what you'll love, than hopefully we can help you enjoy that more, too. It's the software that everyone is using because of these things and more, and the development community has continued to grow during the past two decades. That means help is everywhere. Go Google it.

1.3 Where do I start?

If you haven't downloaded and installed the most recent versions of R and RStudio, you should probably go do that now. We'll wait...

Once you have installed both of these, find and open RStudio on your computer so you can work along with the examples below.

It may be helpful to watch a couple of YouTube videos before going much further, especially if you are stuck already (no shame). There are tons of them out there, including some that walk you through how to install and open R and

RStudio. They range from just a couple of minutes to a couple of hours. Here's one example provided by the How To R Channel.

Depending on how long that took, you may or may not be enthused by the following:

the learning curve for R is steep...like a cliff, not a hill.

But, once you get the hang of it you can learn a lot really quickly. Cheat sheets like these reference cards can help you along the way by serving as miniature reference manuals in the mean-time. There are also *tons* of e-books and websites out there like the one you are reading now. And, there is a huge, active user-community just a Google away. Just searching “how to ____ in R” will return multiple results for most questions, with everything from open-source text books like this to R project websites (e.g. RStan) or programming forums like StackOverflow. You can find links to a few Additional Resources on the course website, but part of learning R is learning how to Google about R.

1.4 Programming conventions

Style and organization

Learning to write code will be easier if you bite the bullet early on and adopt some kind of organization that allows you to interact with it (read, write, run, stare aimlessly, debug) more efficiently.

There are a lot of different ways to write computer code. All of them are intended to increase efficiency and readability. Some rules are more hard-coded and program-specific than others. For example, students in this class will notice that none of my code goes beyond a certain vertical line in the editor. That is to make it so that people don't have to scroll over to the right of the editor to see what I have written when I email them code. When I share code with students I tend to justify everything *really* far to the left because everyone works on tiny laptops with multiple windows open and none of them maximized [shudders].

I suppose there is no “right” way to edit your code, but it will make your life easier if you find a style you like and stick to those conventions. If you are the kind of person who needs order in your life, you can check out the **tidyverse** style guide for tips. You can check code style with the **lintr** package or interactively re-style your code with the **styler** package if you're thinking that may be a lot of work to remember on the front-end.

Regardless of how you end up styling your code, here are a few helpful hints that ought to help you get comfortable with your keyboard. I guess these are probably generally applicable to programming and not specific to R.

Some handy coding tips

Get to know your keyboard and your speed keys for code execution and completion. Use the mouse to navigate the GUI, not to write code. Here is a fairly comprehensive list of speed-key combinations for all of the major operating systems from the Rstudio website. You don't need to know them all, but it can save you a **ton** of time.

File management is wicked important. This is probably one of the primary struggles folks have with starting to learn R and other languages. At the same time, it is a big part of the secret sauce behind good programming. **For this class, I will assume that you are working out of a single working directory (call it something like “quant_bio” or “biol217”). That means I will assume your scripts (.R files) for each chapter are in the same folder on your computer as your the folder that contains your data.**

An example of your class folder might look like this:

Save early and often In general, RStudio is really good about keeping track of things for you, and it is more and more foolproof these days. However, there are still times when it will crash and there is nothing you can do to get your work back unless it has been saved to a file. So, whenever you write code, write it in a source file that is saved in a place you know you can find it. It is the first thing I do when I start a script, and the last thing I do before I run any code.

Please go check out the supplemental materials on the course website or check out the YouTube video linked above for more help getting started in R if you have no idea what I am talking about at this point.

Commenting code is helpful And I will require that you do it, at least to start. Comments are a way for you to explain what your code does and why. This is useful for sharing code or just figuring out what you did six months ago. It could also be that critical piece of clarity that makes me say “Oh, I see what you did there, +1” on your homework.

```
# This is a comment.
# We know because it is preceded
# by a hashtag, or "octothorpe".

# R ignores comments so you have
# a way to write down what you have
# done or what you are doing.
```

Section breaks help organization I like to use the built-in heading style. It works really well for code-folding in R and when I've written a script that is several hundred lines long, sometimes all I want to see are the section headings. Go ahead and type the code below into a source file (File > New File > Rscript

or **Ctrl+Shift+N**) and save it (File > Save As or **Ctrl+S**). Press the little upside-down triangle to the left of the line to see what it does.

```
# Follow a comment with four dashes or hashes  
# to insert a section heading  
  
# Section heading ----  
  
# Also a section heading ####
```

This is really handy for organizing sections in your homework or for breaking code up into smaller sections when you get started. You'll later learn that when you have to do this a lot, there are usually ways you can reduce your code or split it up more efficiently into other files.

Stricter R programming rules

For the next section, open RStudio if it is not already and type the code into a new source file (**Ctrl+Shift+N**).

All code in R is case sensitive. Run the following lines (with the Run button or **Ctrl+Enter**). If you highlight all of them, they will all be run in sequence from top to bottom. Or, you can manually run each line. Running each line can be helpful for learning how to debug code early on.

```
# Same letter, different case  
a <- 1  
A <- 2  
a == A
```

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

So, what just happened? A few things going on here.

1. We've defined a couple of objects for the first time. If we translate the first line of code, we are saying, "Hey R, assign the value of 1 to an object named **a** for me."
2. Note that the two objects are not the same, and R knows this.
3. The **==** that we typed is a logical test that checks to see if the two objects are identical. If they were, then it would have returned a **TRUE** instead of **FALSE**. This **operator** is very useful, and is more or less ubiquitous in object-oriented languages. We will use it extensively for data queries and conditional indexing (ooooh, I know!).

R will overwrite objects sequentially, so don't name two things the same, unless you don't need the first.

```
a <- 1
a <- 2
a # a takes on the second value here
print(a) # This is another way to look at the value of an object
show(a) # And, here is one more
```

Names should be short and meaningful. `a` is a terrible name, even for a temporary object in most cases.

```
myFirstObject <- 1
```

Cheesy, but better...

Punctuation and special symbols are important And, they are annoying to type in names. Avoid them in object names except for underscores “`_`” where you can. I try to stick with lowercase for everything I do except built-in data and data from external files because it is a pain to change everything.

```
myobject <- 1 # Illegible
my.Object <- 1 # Annoying to type
myObject <- 1 # Better, but still annoying
my_object <- 1 # Same: maybe find a less annoying name?
```

Importantly, R doesn't really care and would treat all of these as unique, but equivalent objects in all regards. Worth noting that most R style recommendations are moving toward the last example above.

Some symbol combinations are not allowed in object names But, these are usually bad names or temporary objects that create junk in your workspace anyway.

```
# In Rstudio there are nifty
# little markers to show this
# is broken
# 1a <- 1

# This one works (try it by typing
# "a1" in the console)
a1 <- 1
a2 <- a1 + 1
a3 <- a2 + 1
```


We'll see later that sequential operations that require creation of redundant objects (that require memory) are usually better replaced by over-writing objects in place or using functions like the pipe `%>%` from the `magrittr` package that help us keep a “tidy” workspace.

Some things can be expressed in multiple ways. Both `T` and `TRUE` can be used to indicate a logical that evaluates as being `TRUE`. But `t` is used to transpose data.

```
T == TRUE
```

Some names are “reserved”, “built-in”, or pre-defined. Did you notice that R already knew what `T` and `TRUE` were? We will talk more about this later in the course if we need to.

Other examples include functions like `in`, `if`, `else`, `for`, `function()` and a mess of others have special uses.

Some *symbols* are also reserved for special use as “operators”, like: `+`, `-`, `*`, `% %`, `&`, `/`, `<`, `(`, `{`, `[`, `"`, `'`, `...`, and a bunch of others. We will use basically all of these in just the first couple of chapters.

1.5 Next steps

These are just some basic guidelines that should help you get started working with R and RStudio. In Chapter 2, we will begin working with objects, talk about how R sees those objects, and then look at things we can do to those objects using functions.

Chapter 2

Data structures

Contrast how you see a fish and how computers see fish. Our job is to bridge the gap. No problem...

In this chapter, we will introduce basic data structures and how to work with them in R. One of our challenges is to understand how R sees our data.

R is what is known as a “high-level” or “interpreted” programming language, in addition to being “functional” and “object-oriented”. This means the pieces that make it up are a little more intuitive to the average user than most low-level languages like C or C++. The back-end of R is, in fact, a collection of low-level code that builds up the functionality that we need. This means that R has a broad range of uses, from data management to math, and even GIS and data visualization tools, all of which are conveniently wrapped in an “intuitive”, “user-friendly” language.

Part of this flexibility comes from the fact that R is also a “vectorized” language. Holy cow, R is so many things. But, why do you care about this? This will help you wrap your head around how objects are created and stored in R, which will help you understand how to make, access, modify, and combine the data that you will need for any approach to data analysis. It is maybe easiest to see by taking a look at some of the data structures that we’ll work with.

We will work exclusively with objects and functions created in base R for this Chapter, so you do not need any of the class data sets to play along.

2.1 Vectors

The vector is the basic unit of information in R. Pretty much everything else we’ll concern ourselves with is made of vectors and can be contained within one. Wow, what an existential paradox *that* is.

Let's take a look at how this works and why it matters. Here, we have defined **a** as a variable with the value of 1.

```
a <- 1
```

...or have we?

```
print(a)
```

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

What is the square bracket in the output here? It's an index. The index is telling us that the first element of **a** is 1. This means that **a** is actually a “vector”, not a “scalar” or singular value as you may have been thinking about it. You can think of a vector as a column in an Excel spreadsheet or an analogous data table. By treating every object (loosely) as a vector, or an element thereof, the language becomes much more general.

So, even if we define something with a single value, it is still just a vector with one element. For us, this is important because of the way that it lets us do math. It makes vector operations so easy that we don't even need to think about them when we start to make statistical models. It makes working through the math a zillion times easier than on paper! In terms of programming, it can make a lot of things easier, too.

An **atomic vector** is a vector that can hold one and only one kind of data. These can include:

- Character
- Numeric
- Integer
- Logical
- Factor
- Date/time

And some others, but none with which we'll concern ourselves here.

If you are ever curious about what kind of object you are working with, you can find out by exposing the data structure with **str()**:

Let's go play with some!

```
str(a)
```

```
##  num 1
```

Examples of atomic vectors follow. Run the code to see what it does:

Integers and numerics

First, we demonstrate one way to make a vector in R. The `c()` function (“combine”) is our friend here for the quick-and-dirty approach.

In this case, we are making an object that contains a sequence of whole numbers, or integers.

```
# Make a vector of integers 1-5
a <- c(1, 2, 3, 4, 5)

# One way to look at our vector
print(a)
```

Here is another way to make the same vector, but we need to pay attention to how R sees the data type. A closer look shows that these methods produce a **numeric** vector (`num`) instead of an **integer** vector (`int`). For the most part, this one won’t make a huge difference, but it can become important when writing or debugging statistical models.

```
# Define the same vector using a sequence
a <- seq(from = 1, to = 5, by = 1)
str(a)
```

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

We can change this by explicitly telling R how to build our vector:

```
a <- as.vector(x = seq(1, 5, 1), mode = "numeric")
```

Notice that I did not include the argument names in the call to `seq()` because these are commonly used default arguments.

Characters and factors

Characters are anything that is represented as text strings.

```
b <- c("a", "b", "c", "d", "e") # Make a character vector
b # Print it to the console
```

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

```
str(b) # Now it's a character vector
```

```
## chr [1:5] "a" "b" "c" "d" "e"
```

They are readily converted (sometimes automatically) to **factors**:

```
b <- as.factor(b) # But we can change if we want
b
```

```
## [1] a b c d e
## Levels: a b c d e
```

```
str(b) # Look at the data structure
```

```
## Factor w/ 5 levels "a","b","c","d",...: 1 2 3 4 5
```

Factors are a special kind of data type in R that we may run across from time to time. They have **levels** that can be ordered numerically. This is not important except that it becomes useful for coding variables used in statistical models- R does most of this behind the scenes and we won't have to worry about it for the most part. In fact, in a lot of cases we will want to change factors to numerics or characters so they are easier to manipulate.

This is what it looks like when we code a factor as number:

```
as.numeric(b)
```

```
# What did that do?
?as.numeric
```

Aside: we can ask R what functions mean by adding a question mark as we do above. And not just functions: we can ask it about pretty much any built-in object. The help pages take a little getting used to, but once you get the hang of it... In the mean time, the internet is your friend and you will find a multitude of online groups and forums with a quick search.

Logical vectors

Most of the **logical** vectors we deal with are yes/no or comparisons to determine whether a given piece of information matches a condition. Here, we use a logical check to see if the object **a** we created earlier is the same as object **b**. If we store the results of this check to a new object **c**, we get a new logical vector filled with **TRUE** and **FALSE**, one for each element in **a** and **b**.

```
# The "==" compares the numeric vector to the factor one
c <- a == b
c
```

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

```
str(c)
```

```
## logi [1:5] FALSE FALSE FALSE FALSE FALSE
```

We now have a logical vector. For the sake of demonstration, we could perform any number of logical checks on a vector using built-in R functions (it does not need to be a logical like `c` above).

We can check for missing values.

```
is.na(a)
```

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

We can make sure that all values are finite.

```
is.finite(a)
```

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

The exclamation ! point means “not” in to computers.

```
!is.na(a)
```

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

We can see if specific elements meet a criterion.

```
a == 3
```

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

We can just look at unique values.

```
unique(b)
```

```
## [1] a b c d e  
## Levels: a b c d e
```

The examples above are all simple vector operations. These form the basis for data manipulation and analysis in R.

2.2 Vector operations

A lot of data manipulation in R is based on logical checks like the ones shown above. We can take these one step further to actually perform what one might think of as a query.

For example, we can reference specific elements of vectors directly. Here, we specify that we want to print the third element of `a`.

```
# This one just prints it  
a[3]
```

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

We might want to store that value to a new object `f` that is easier to read and type out.

```
# This one stores it in a new object  
f <- a[3]
```

Important

If it is not yet obvious, we have to assign the output of functions to new objects for the values to be usable in the future. In the example above, `a` is never actually *changed*. This is a common source of confusion early on.

Going further, we could select vector elements based on some condition. On the first line of code, we tell R to show us the indices of the elements in vector `b` that match the character string `c`. Out loud, we would say, “`b` where the value of `b` is equal to `c`” in the first example. We can also use built-in R functions to just store the indices for all elements of `b` where `b` is equal to the character string “`c`”.


```
b[b == "c"]
```

```
## [1] c
## Levels: a b c d e
```

```
which(b == "c")
```

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

Perhaps more practically speaking, we can do elementwise operations on vectors easily in R. Here are a bunch of different things that you might be interested in doing with the objects that we've created so far. Give a few of these a try. Perhaps more practically speaking, we can do elementwise operations on vectors easily in R. Give a few of these a shot.

```
a * .5 # Multiplication
a + 100 # Addition
a - 3 # Subtraction
a / 2 # Division
a^2 # Exponentiation
exp(a) # Same as "e to the a"
log(a) # Natural logarithm
log10(a) # Log base 10
```

If we change `b` to `character`, we can do string manipulation, too!

```
# Convert b to character
b <- as.character(b)
```

We can append text. Remember, the examples below will just print the result. We would have to overwrite `b` or save it to a new object if we wanted to be able to use the result somewhere else later.

```
# Paste an arbitrary string on to b
paste(b, "AAAA", sep = "")
```

```
## [1] "aAAAA" "bAAAA" "cAAAA" "dAAAA" "eAAAA"
```

```
# We can do it the other way
paste("AAAA", b, sep = "")
```

```
## [1] "AAAAa" "AAAAb" "AAAAc" "AAAAd" "AAAAe"
```

```
# Add symbols to separate
paste("AAAA", b, sep = "--")

## [1] "AAAA--a" "AAAA--b" "AAAA--c" "AAAA--d" "AAAA--e"
```

```
# We can replace text
gsub(pattern = "c", replacement = "AAAA", b)
```

```
## [1] "a"      "b"      "AAAA" "d"      "e"
```

```
# Make a new object
e <- paste("AAAA", b, sep = "")

# Print to console
e
```

```
## [1] "AAAAa" "AAAAb" "AAAAc" "AAAAd" "AAAAe"
```

```
# We can strip text
# (or dates, or times, etc.)
substr(e, start = 5, stop = 5)
```

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

We can check how many elements are in a vector.

```
# A has a length of 5,
# try it and check it
length(a)
```

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

```
# Yup, looks about right
a
```

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

And we can do lots of other nifty things like this. We can also bind multiple vectors together into a rectangular **matrix**. Say what?

2.3 Matrices

Matrices are rectangular objects that we can think of as being made up of vectors.

We can make matrices by binding vectors that already exist.

```
cbind(a, e)
```

```
##      a    e
## [1,] "1" "AAAAa"
## [2,] "2" "AAAAb"
## [3,] "3" "AAAAc"
## [4,] "4" "AAAAd"
## [5,] "5" "AAAAe"
```

Or we can make an empty one to fill.

```
matrix(0, nrow = 3, ncol = 4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
```

Or we can make one from scratch.

```
mat <- matrix(seq(1, 12), ncol = 3, nrow = 4)
```

We can do all of the things we did with vectors to matrices, but now we have more than one column, and official “rows” that we can also use to these ends:

```
ncol(mat) # Number of columns
nrow(mat) # Number of rows
length(mat) # Total number of entries
mat[2, 3] # Value of row 2, column 3
str(mat)
```

See how number of rows and columns is defined in data structure? With rows and columns, we can assign column names and row names.

```
colnames(mat) <- c("first", "second", "third")
rownames(mat) <- c("This", "is", "a", "matrix")

# Take a look
mat
```

```
##           first second third
## This         1      5     9
## is           2      6    10
## a            3      7    11
## matrix       4      8    12
```

We can also do math on matrices just like vectors, because matrices are just vectors smooshed into two dimensions (it's totally a word).

```
mat * 2
```

```
##           first second third
## This         2     10    18
## is           4     12    20
## a            6     14    22
## matrix       8     16    24
```

All the same operations we did on vectors above...one example.

More on matrices as we need them. We won't use these a lot in this module, but R relies heavily on matrices to do linear algebra behind the scenes in the models that we will be working with.

2.4 Dataframes

Dataframes are like matrices, only not. They have a row/column structure like matrices and are also rectangular in nature. But, they can hold more than one data type!

Dataframes are made up of atomic vectors.

This is probably the data structure that we will use most in this book, along with atomic vectors.

Let's make a dataframe to see how it works.

```
# Make a new object 'a' from a sequence
a <- seq(from = .5, to = 10, by = .5)

# Vector math: raise each 'a' to power of 2
b <- a^2

# Replicates values in object a # of times
c <- rep(c("a", "b", "c", "d"), 5)

# Note, we don't use quotes for objects,
# but we do for character variables
d <- data.frame(a, b, c)
```

Now we can look at it:

```
print(d)
```

```
##      a      b c
## 1  0.5  0.25 a
## 2  1.0  1.00 b
## 3  1.5  2.25 c
## 4  2.0  4.00 d
## 5  2.5  6.25 a
## 6  3.0  9.00 b
## 7  3.5 12.25 c
## 8  4.0 16.00 d
## 9  4.5 20.25 a
##10  5.0 25.00 b
##11  5.5 30.25 c
##12  6.0 36.00 d
##13  6.5 42.25 a
##14  7.0 49.00 b
##15  7.5 56.25 c
##16  8.0 64.00 d
##17  8.5 72.25 a
##18  9.0 81.00 b
##19  9.5 90.25 c
##20 10.0 100.00 d
```

Notice that R assigns names to dataframes on the fly based on object names that you used to create them unless you specify elements of a data frame like this. They are not `colnames` as with matrices, they are `names`. You can set them when you make the dataframe like this:

```
d <- data.frame(a = a, b = b, c = c)
```

Now can look at the names.

```
# All of the names
names(d)
```

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

```
# One at a time: note indexing, names(d) is a vector!!
names(d)[2]
```

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

We can change the names.

```
# All at once- note quotes
names(d) <- c("Increment", "Squared", "Class")

# Print it to see what this does
names(d)

# Or, change one at a time..
names(d)[3] <- "Letter"

# Print it again to see what changed
names(d)
```

We can also rename the entire dataframe.

```
e <- d
```

Have a look:

```
# Head shows first six
# rows by default
head(e)
```

```
##      a      b c
## 1 0.5 0.25 a
## 2 1.0 1.00 b
## 3 1.5 2.25 c
## 4 2.0 4.00 d
## 5 2.5 6.25 a
## 6 3.0 9.00 b
```

```
# Or, we can look at any  
# other number that we want  
head(e, 10)
```

```
##      a      b c  
## 1  0.5  0.25 a  
## 2  1.0  1.00 b  
## 3  1.5  2.25 c  
## 4  2.0  4.00 d  
## 5  2.5  6.25 a  
## 6  3.0  9.00 b  
## 7  3.5 12.25 c  
## 8  4.0 16.00 d  
## 9  4.5 20.25 a  
## 10 5.0 25.00 b
```

We can make new columns in data frames like this!

```
# Make a new column with the  
# square root of our increment  
# column  
e$Sqrt <- sqrt(e$Increment)  
e
```

Looking at specific elements of a dataframe is similar to a matrix, with some added capabilities. We'll do this with a real data set so it's more fun. There are a whole bunch of built-in data sets that we can use for examples. Let's start by looking at the `iris` data.

```
# This is how you load built-in  
# data sets  
data("iris")
```

Play with the functions below to explore how this data set is stored in the environment, and how R sees it. This is a good practice to get into in general.

```
# We can use ls() to see  
# what is in our environment  
ls()  
  
# Look at the first six rows  
# of data in the object  
head(iris)
```

```
# How many rows does it have?
nrow(iris)

# How many columns?
ncol(iris)

# What are the column names?
names(iris)

# Have a look at the data structure-
# tells us all of the above
str(iris)

# Summarize the variables
# in the dataframe
summary(iris)
```

Now let's look at some specific things.

```
# What is the value in 12th row
# of the 4th column of iris?
iris[12, 4]
```

```
## [1] 0.2
```

```
# What is the mean sepal length
# among all species in iris?
mean(iris$Sepal.Length)
```

```
## [1] 5.843333
```

What about the mean of `Sepal.Length` just for `setosa`?

A couple of new things going on here:

1. We can refer to the columns as atomic vectors within the dataframe if we want to. Some times we have to do this...
2. Note the logical check for species

What we are saying here is, “Hey R, show me the mean of the column `Sepal.Length` in the dataframe `iris` where the species name is `setosa`”


```
mean(iris$Sepal.Length[iris$Species == "setosa"])
```

```
## [1] 5.006
```

We can write this out longhand to make sure it's correct (it is).

```
logicalCheck <- iris$Species == "setosa"
lengthCheck <- iris$Sepal.Length[iris$Species == "setosa"]
```

We can also look at the whole data frame just for `setosa`.

```
# Note that the structure of species
# is preserved as a factor with three
# levels even though setosa is the
# only species name in the new df
setosaData <- iris[iris$Species == "setosa", ]

str(setosaData)
```

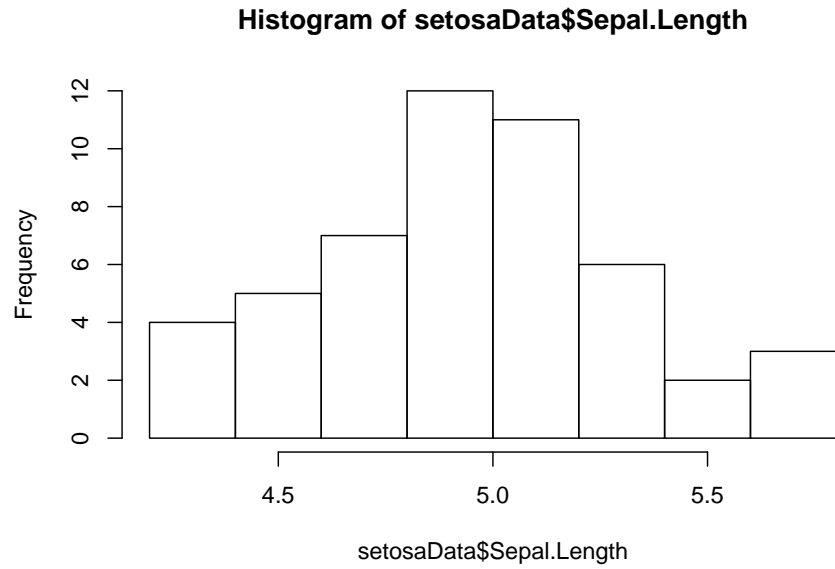
```
## 'data.frame': 50 obs. of 5 variables:
## $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
## $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
## $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
## $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
## $ Species : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

Finally, once we are working with dataframes, plotting becomes much easier to understand, and we can ease into some rudimentary, clunky R plots.

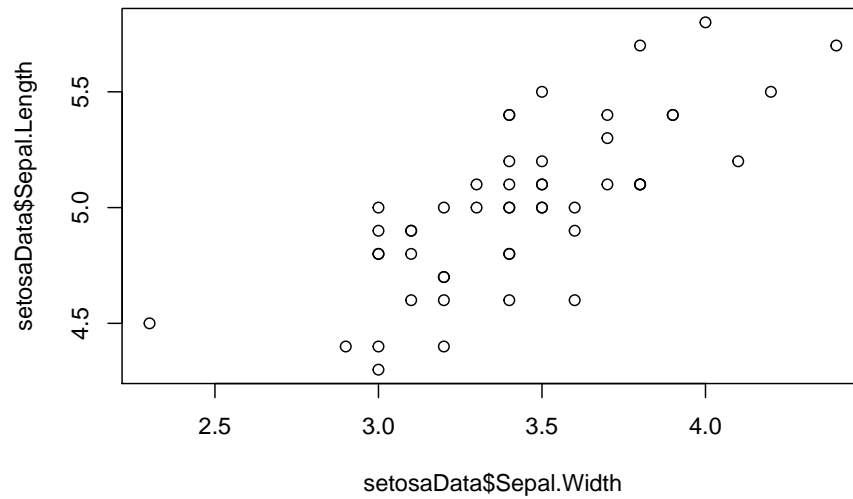
```
# Some quick plotting code

# Once we have a nice dataframe like
# these ones, we can actually step into
# The world of exploratory analyses.

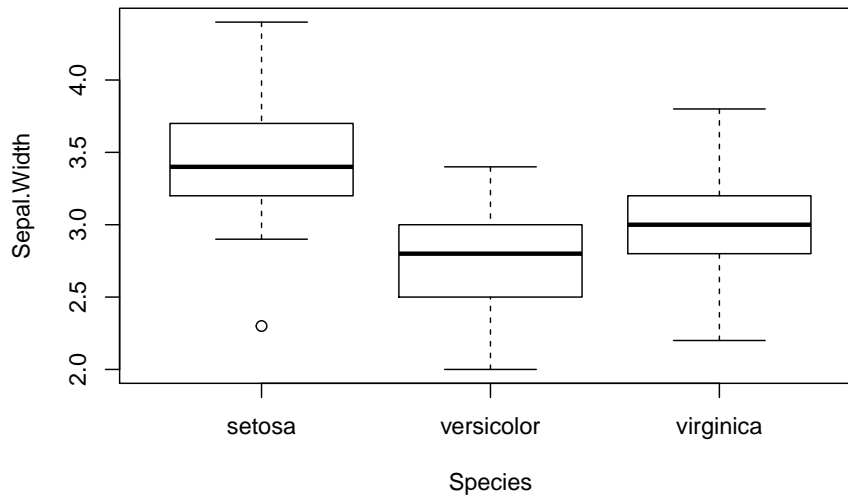
# Make a histogram of sepal lengths
hist(setosaData$Sepal.Length)
```



```
# Bi-plot  
plot(setosaData$Sepal.Width, setosaData$Sepal.Length)
```



```
# Boxplots  
boxplot(Sepal.Width ~ Species, data = iris)
```



Much, **MUCH** more of this to come as we continue.

2.5 Lists

Lists are the ultimate data type in R. They are actually a vector that can hold different kinds of data, like a dataframe. In fact, a dataframe is just a spectacularly rectangular list. Each element of a list can be any kind of object (an atomic vector, a matrix, a dataframe, or even another list!!).

Most of the real, filthy R programming relies heavily on lists. We will have to work with them at some point in this class, but we won't take a ton of time on them here.

Let's make a list- just to see how they work. Notice how our index operator has changed from `[]` to `[[]]`? And, at the highest level of organization, we have only one dimension in our list, but any given element `myList[[i]]` could hold any number of dimensions.

```
# Create an empty list with four elements  
myList <- vector(mode = "list", length = 4)
```

```
# Assign some of our previously
# created objects to the elements
myList[[1]] <- a
myList[[2]] <- c
myList[[3]] <- mat
myList[[4]] <- d
```

Have a look at the list:

```
# Print it
# Cool, huh?
myList

## [[1]]
## [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5
## [16] 8.0 8.5 9.0 9.5 10.0
##
## [[2]]
## [1] "a" "b" "c" "d" "a" "b" "c" "d" "a" "b" "c" "d" "a" "b" "c" "d" "a" "b" "c"
## [20] "d"
##
## [[3]]
##           first second third
## This           1      5      9
## is              2      6     10
## a               3      7     11
## matrix          4      8     12
##
## [[4]]
##           a           b c
## 1  0.5    0.25 a
## 2  1.0    1.00 b
## 3  1.5    2.25 c
## 4  2.0    4.00 d
## 5  2.5    6.25 a
## 6  3.0    9.00 b
## 7  3.5   12.25 c
## 8  4.0   16.00 d
## 9  4.5   20.25 a
## 10 5.0   25.00 b
## 11 5.5   30.25 c
## 12 6.0   36.00 d
## 13 6.5   42.25 a
## 14 7.0   49.00 b
## 15 7.5   56.25 c
```

```
## 16  8.0  64.00 d
## 17  8.5  72.25 a
## 18  9.0  81.00 b
## 19  9.5  90.25 c
## 20 10.0 100.00 d
```

You can assign names when you create the list like we did for dataframes, too. You can do this manually, or R will do it on the fly for you. You can also reassign names to a list that you've already created.

```
# No names by default
names(myList)

# Give it names like we did with
# a dataframe
names(myList) <- c("a", "c", "mat", "d")

# See how the names work now?
myList

# We reference these differently [[]]
myList[[1]]

# But we can still get into each object
# Play around with the numbers to see what they do!
myList[[2]][5]

# Can also reference it this way!
myList$c[1]
```

Very commonly, model objects and output are stored as lists. In fact, most objects that require a large amount of diverse information in R pack it all together in one place using lists, that way we always know where to find it and how as long as the objects are documented. Conceptually, every object in R, from your workspace on down the line, is a list **AND** an element of a list. It seems like a lot to take in now, but will be very useful in the future.

2.6 Next steps

For more practice with the data structures and R functions we covered here, you can check out this walk-through of basic R commands from the How To R YouTube Channel.

In the Chapter 3(#Chapter3), we will begin using functions from external R packages to read and work with real data.

Chapter 3

Working with data

American shad, the best fish, lost in the data deluge. Let's figure out how to make some sense of it.

The purpose of this chapter is to get you comfortable working with data in R and give you some tools for summarizing those data in a meaningful way. This is not meant to be a comprehensive treatment of these subjects but rather an introduction to the tools that are available to you (say it with me: “Worst Stats Text eveR”). There are a lot of tools out there and you may come up with something that works better for you once you have some basics under your belt.

Now that you have a handle on the types of data you can expect to run into in R, let's have a look at how we read and work with data that we get from the real world.

We will work with the `ctr_fish.csv` file for Chapter 3, so you will need to download the class data sets that go with this book to play along. We will also need the **tidyverse** package, but instructions for installation are provided below because this is the first time we have downloaded and installed a package.

3.1 Data read

There are few things that will turn someone away from a statistical software program faster than if they can't even figure out how to get the program to read in their data. So, we are going to get it out of the way right up front!

Let's start by reading in a data file - this time we use real data.

The data are stored in a “comma separated values” file (`.csv` extension). This is a fairly universal format, so we read it in using the fairly universal `read.csv()` function. This would change depending on how the data were stored, or how

big the data files were, but that is a topic further investigation for a later date. I probably do 95% of my data reads using `.csv` files. We'll look at a few others later.

Important Remember that I am assuming your scripts are in the same directory (folder) on your computer as where you downloaded and unzipped the class data (see here for reminder).

Before you can read this file you will need to set your working directory. For class, I will ask that you click **Session > Set Working Directory > To Source File Location**. This will set the working directory to wherever you have saved your code so that R can find the folder `data` and the files inside of it. You'll notice that R spits out some code in the console when you click this. You can also use that code to set a working directory in your script but that can cause all kinds of problems, so don't do it.

```
# Start by reading in the data
am_shad <- read.csv("data/ctr_fish.csv")
```

Once you've read your data in, it's always a good idea to look at the first few lines of data to make sure nothing looks 'fishy'. Ha-ha, I couldn't help myself!

These are sex-specific length and age data for American shad (*Alosa sapidissima*) from the Connecticut River, USA. The data are used in models that I maintain with collaborators from NOAA Fisheries, the US Geological Survey, the US Fish and Wildlife Service, and others. The data were provided by CT Department of Energy and Environmental Protection (CTDEEP) and come from adult fish that return to the river from the ocean each year to spawn in fresh water.

You can look at the first few rows of data with the `head()` function:

```
# Look at the first 10 rows
head(am_shad, 10)
```

##	Sex	Age	Length	yearCollected	backCalculated	Mass
## 1	B	1	13	2010	TRUE	NA
## 2	B	1	15	2010	TRUE	NA
## 3	B	1	15	2010	TRUE	NA
## 4	B	1	15	2010	TRUE	NA
## 5	B	1	15	2010	TRUE	NA
## 6	B	1	15	2010	TRUE	NA
## 7	B	1	16	2010	TRUE	NA
## 8	B	1	16	2010	TRUE	NA
## 9	B	1	16	2010	TRUE	NA
## 10	B	1	16	2010	TRUE	NA

The NA values are supposed to be there. They are missing data.

And, don't forget about your old friend `str()` for a peek at how R sees your data. This can take care of a lot of potential problems later on.

```
# Look at the structure of the data
str(am_shad)
```

```
## 'data.frame': 16946 obs. of  6 variables:
##  $ Sex          : Factor w/ 2 levels "B","R": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Age           : int   1 1 1 1 1 1 1 1 1 1 ...
##  $ Length        : num   13 15 15 15 15 15 16 16 16 16 ...
##  $ yearCollected: int   2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 ...
##  $ backCalculated: logi   TRUE TRUE TRUE TRUE TRUE TRUE ...
##  $ Mass          : int   NA NA NA NA NA NA NA NA NA NA ...
```

There are about 17,000 observations (rows) of 6 variables (columns) in this data set. Here is a quick breakdown:

Sex: fish gender. B stands for 'buck' (males), R stands for 'roe' (females).

Age: an integer describing fish age.

Length: fish length at age (cm).

yearCollected: the year in which the fish was caught.

backCalculated: a logical indicating whether or not the length of the fish was back-calculated from aging.

Mass: the mass of individual fish (in grams). Note that this is NA for all ages that were estimated from hard structures (so all cases for which `backCalculated == TRUE`).

3.2 Quick data summaries

There are a number of simple ways to summarize data quickly in base R. We already looked at a few of these in previous chapters. But what about something a little more in-depth?

One quick way to look at your data is using the `summary()` function

```
summary(am_shad)
```

```
## Sex           Age           Length      yearCollected  backCalculated
## B:9512   Min.    :1.000   Min.    : 3.00   Min.    :2010   Mode :logical
## R:7434   1st Qu.:2.000   1st Qu.:31.00   1st Qu.:2011   FALSE:3046
##           Median :3.000   Median :38.00   Median :2012   TRUE :13900
##           Mean   :3.155   Mean   :36.39   Mean   :2012
##           3rd Qu.:4.000   3rd Qu.:43.00   3rd Qu.:2013
##           Max.    :7.000   Max.    :55.00   Max.    :2014
##
##           Mass
## Min.    : 0
## 1st Qu.: 900
## Median :1120
## Mean   :1173
## 3rd Qu.:1440
## Max.    :3280
## NA's    :14115
```

This is useful for getting the big-picture. For continuous variables (e.g., `Age` and `Length`) R will report some descriptive statistics like the `mean`, `median`, and quantiles. For discrete variables (e.g. `Sex` and `backCalculated`) we get the mode (if not `factor` or `chr`) and counts of observations within each discrete level (e.g. number of observations of B and R in the variable `Sex`).

But, this approach doesn't really give us much info.

We can create more meaningful summaries pretty easily if we install and load some packages like we talked about in Chapter 1, and then look at different ways of sub-setting the data with base R and some methods that might be a little more intuitive for you.

3.3 Subsetting and selecting data

Before we can make meaningful data summaries, we will probably need to re-organize our data in a logical way (through sub-setting or selected specific chunks of data). A lot of times, we do this along the way without really thinking about it.

3.3.1 Manual subsets and selections

We talked a little about sub-setting data with logical queries in Chapter 2. Now, let's refresh and take that a little further to see why we might want to do that.

First, we'll select just the data from `am_shad` where `backCalculated` was `FALSE`. This will give us only the measured `Length` and `Mass` for each of the fish, along with their `Sex` and `yearCollected`. I'll call this new object `measured`. Remember, `am_shad` is a data frame, so it has two dimensions when we use `[]` for sub-setting and these are separated by a comma, like this: `object[rows, columns]`. When we leave the columns blank, R knows that it should keep all of the columns.

```
measured <- am_shad[am_shad$backCalculated == FALSE, ]
```

We could do this for as many conceivable conditions in our data on which we may wish to subset, but the code can get clunky and hard to manage. For example can you imagine re-writing this if you just want to select age six roes without back-calculated lengths?

```
# Notice how we string together multiple
# conditions with "&". If these were 'or'
# we would use the vertical pipe "|"
age_6_rows_measured <- am_shad[am_shad$backCalculated == FALSE &
                                am_shad$Sex == "R" &
                                am_shad$Age == 6, ]
```

3.3.2 Subsetting and summaries in base R

This notation can be really confusing to folks who are just trying to learn a new programming language. Because of that, there are great functions like `subset()` available that are more intuitive (but less clear to programmers). You could also subset the data using the following code:

```
measured <- subset(am_shad, backCalculated == FALSE)
```

We could also get our age-six females from the previous example using this approach, and at least the code is a little cleaner:

```
age_6_roes_measured <- subset(am_shad,
                              backCalculated == FALSE &
                              Sex == "R" &
                              Age == 6
                              )
```

Both do the same thing, but we'll see later that using `subset` is preferable if we plan on chaining together a bunch of data manipulation commands using pipes (`%>%`).

Next, we might be interested to know how many fish we have represented in each **Sex**. We can find this out using the `table` function in base R:

```
# Here, I use the column name because
# we just want all observations of a single
# variable. Be careful switching between names,
# numbers, and $names!
table(measured['Sex'])
```

```
##
##      B      R
## 1793 1253
```

We see that we have 1793 females and 1253 males.

We can also get tallies of the number of fish in each **Age** for each **Sex** if we would like to see that:

```
table(measured$Sex, measured$Age)
```

```
##
##           3    4    5    6    7
##  B 255 848 579 108    3
##  R   0 361 658 220   14
```

But, what if we wanted to calculate some kind of summary statistic, like a **mean** and report that by group?

For our age-6 females example, it would look like this:

```
age_6_roes_measured <- subset(am_shad,
                             backCalculated == FALSE &
                             Sex == "R" &
                             Age == 6
                             )
age_6_female_mean <- mean(age_6_roes_measured$Length)
```

Again, we could do this manually, but would require a lot of code for a simple calculation if we use the methods above all by themselves to get these for each age group of roes.

We would basically just copy-and-paste the code over and over to force R into making the data summaries we need. Nothing wrong with this approach, and it certainly has its uses for simple summaries, but it can be cumbersome and

redundant. It also fills your workspace up with tons of objects that are hard to keep track of and that will cause your code-completion suggestions to be *wicked* annoying in RStudio.

That usually means there is a better way to write the code...

3.3.3 Subsetting and summaries in the tidyverse

Long ago, when I was still a noOb writing R code with a stand-alone text editor and a console there were not a ton of packages available for the express purpose of cleaning up data manipulation in R. The one I relied on most heavily was the `plyr` package. Since then, R has grown and a lot of these functions have been gathered under the umbrella of the tidyverse, which is a collection of specific R packages designed to make the whole process less painful. These include packages like `dplyr` (which replaced `plyr`) and others that are designed to work together with similar syntax to make data science (for us, data manipulation and presentation) a lot cleaner and better standardized. We will rely heavily on packages in the tidyverse throughout this book.

Before we can work with these packages, however, we need to install them - something we haven't talked about yet! Most of the critical R packages are hosted through the Comprehensive R Archive Network, or CRAN. Still, tons of others are available for installation from hosting services like GitHub and GitLab.

If you haven't seen it yet, here is a three-minute video explaining how to install packages using RStudio. **Watch it. Please.**

It is also easy to install packages by running a line of code in the console. We could install each of the packages in the tidyverse separately. But we can also get all of them at once because they are all packaged together, too.

Follow the instructions in the YouTube link above, or install the package from the command line:

```
install.packages('tidyverse')
```

Once we have installed these packages, we can use the functions in them to clean up our data manipulation pipeline and get some really useful information.

3.4 Better data summaries

Now, we'll look at some slightly more advanced summaries. Start by loading the `dplyr` package into your R session with the following code.

```
library(dplyr)
```

We can use functions from the `dplyr` package to calculate mean `Length` of fish for each combination of `Sex` and `Age` group much more easily than we did for a single group above.

First, we group the data in `measured` data frame that we created previously using the `group_by` function. For this, we just need to give R the data frame and the variables by which we would like to group:

```
g_lengths <- group_by(measured, Sex, Age)
```

This doesn't change how we see the data much (it gets converted to a `tibble`), just how R sees it.

Next, we summarize the variable `Length` by `Sex` and `Age` using the `summarize` function:

```
sum_out <- summarize(g_lengths, avg = mean(Length))

head(sum_out)
```

```
## # A tibble: 6 x 3
## # Groups:   Sex [2]
##   Sex      Age  avg
##   <fct> <int> <dbl>
## 1 B         3  38.1
## 2 B         4  40.5
## 3 B         5  42.0
## 4 B         6  43.4
## 5 B         7  46.8
## 6 R         4  45.0
```

Wow! That was super-easy!

Finally, to make things even more streamlined, we can chain all of these operations together using the `%>%` function from `magrittr`. This really cleans up the code and gives us small chunks of code that are easier to read than the dozens of lines of code it would take to do this manually.

```
# This will do it all at once!
sum_out <- # Front-end object assignment
  measured %>% # Pass measured to the group_by function
  group_by(Sex, Age) %>% # Group by Sex and age and pass to summarize
  summarize(avg = mean(Length))
```

We could also assign the output to a variable at the end, whichever is easier for you to read:

```
measured %>% # Pass measured to the group_by function
group_by(Sex, Age) %>% # Group by Sex and age and pass to summarize
summarize(avg = mean(Length)) -> sim_out # Back-end object assignment
```

And, it is really easy to get multiple summaries out like this at once:

```
sum_out <-
  measured %>%
  group_by(Sex, Age) %>%
  summarize(avg = mean(Length), s.d. = sd(Length))

head(sum_out)
```

```
## # A tibble: 6 x 4
## # Groups:   Sex [2]
##   Sex      Age  avg  s.d.
##   <fct> <int> <dbl> <dbl>
## 1 B         3  38.1  2.75
## 2 B         4  40.5  2.70
## 3 B         5  42.0  2.29
## 4 B         6  43.4  2.09
## 5 B         7  46.8  1.61
## 6 R         4  45.0  2.65
```

Isn't that slick? Just think how long that would have taken most of us in Excel!

This is just one example of how functions in packages can make your life easier and your code more efficient. Now that we have the basics under our belts, let's move on to how we create new variables.

3.5 Creating new variables

There are basically two ways to create new variables: we can modify an existing variable (groups or formulas), or we can simulate new values for that variable (random sampling.)

If we have a formula that relates two variables, we could predict one based on the other deterministically.

For example, I have fit a length-weight regression to explain the relationship between `Length` and `Mass` using the `am_shad` data we've worked with in previous sections.

This relationship looks like your old friend $y = mx + b$, the equation for a line, but we \log_{10} -transform both of the variables before fitting the line (more to come later in the class). Using this relationship, we can predict our **independent variable** (Mass) from our **dependent variable** (Length) if we plug in new values for Length and the **parameters** of the line.

In this case, I know that $m = 3.0703621$, and $b = -1.9535405$.

If I plug these numbers in to the equation above, I can predict $\log_{10}(\text{Mass})$ for new lengths $\log_{10}(\text{Length})$:

$$\log_{10} \text{Mass} = 3.0703621 \cdot \log_{10} \text{Length} - 1.9535405$$

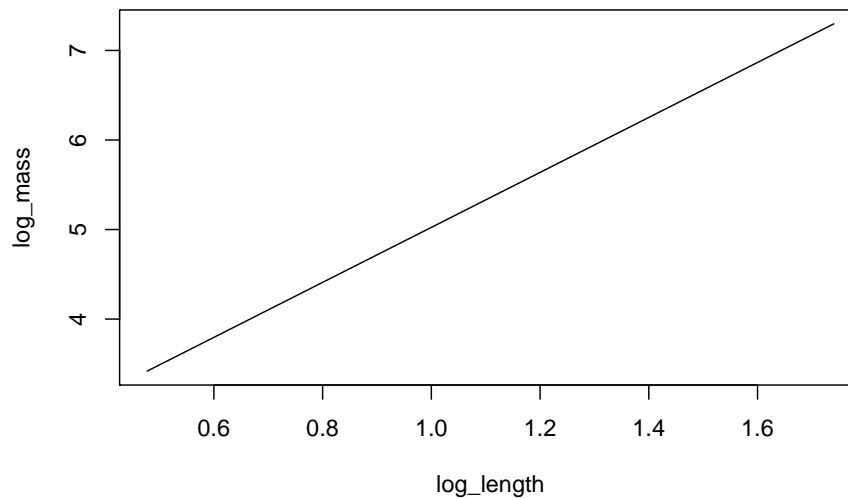
In R, this looks like:

```
# Parameters from length-weight regression
m <- 3.0703621
b <- 1.9535405

# Make a sequence of new lengths based on range in data,
# then take the log of the whole thing all at once.
log_length <- log10( seq(min(am_shad$Length), max(am_shad$Length), 1) )

# Calculate a new thing (log10_mass) using parameters for line
# and sequence of new log10_length.
log_mass <- m * log_length + b

# Plot the prediction
plot(x = log_length, y = log_mass, type = "l")
```

3.6 Data simulation

The point of simulation is usually to account for uncertainty in some process (i.e. we could just pick a single value if we knew it). This is almost always done based on probability. There are a number of ways we could do this. One is by drawing from some probability distribution that we have described, and the other is by randomly sampling data that we already have.

3.6.1 Random sub-samples from a dataset

Let's say we want to take random samples from our huge data set so we can fit models to a subset of data and then use the rest of our data for model validation in weeks to come.

We have around 17,000 observations in the `am_shad` data set. But, what if we wanted to know what it would look like if we only had 100 samples from the same population?

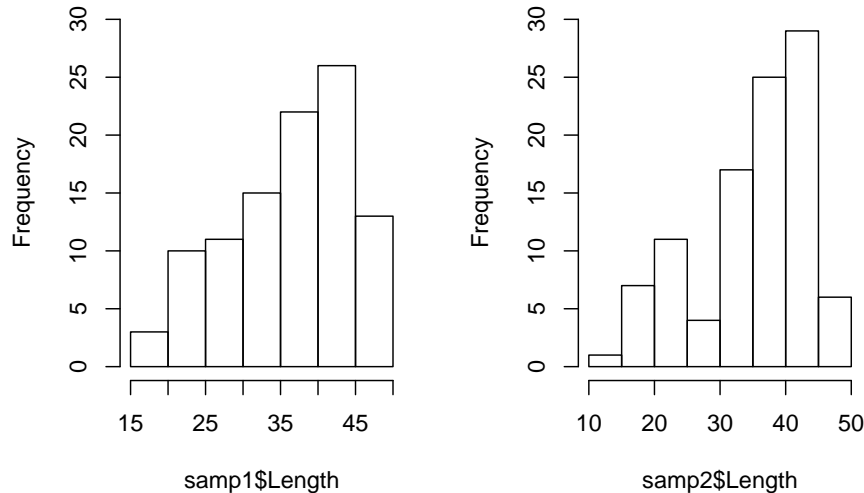
First, tell R how many samples you want.

```
n_samples <- 100
```

Now let's take two samples of 100 fish from our dataframe to see how they compare:

```
# Randomly sample 100 rows of data from our data frame two different
# times to see the differences
samp1 <- am_shad[sample(nrow(am_shad), size = n_samples, replace = FALSE), ]
samp2 <- am_shad[sample(nrow(am_shad), size = n_samples, replace = FALSE), ]

# We can look at them with our histograms
par(mfrow = c(1, 2))
hist(samp1$Length, main = "", ylim = c(0, 30))
hist(samp2$Length, main = "", ylim = c(0, 30))
```



*If you are struggling to get your plotting window back to “normal” after this, you can either click the broom button in your “Plots” window, or you can run the following code for now:

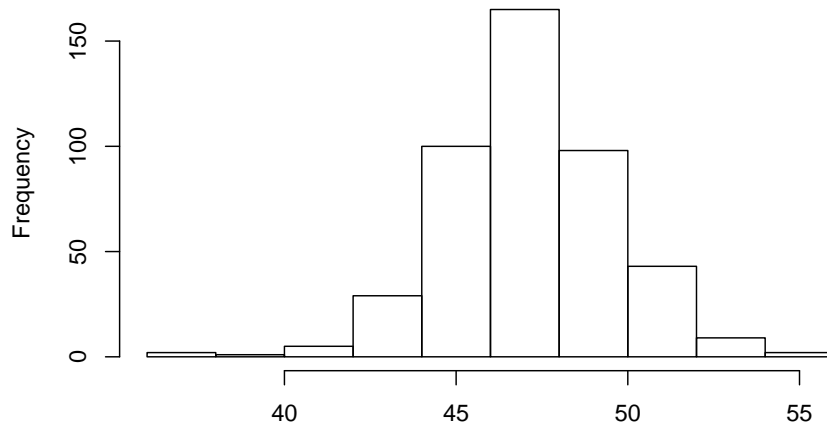
3.6.2 Stochastic simulation

Now, instead of sampling our data let’s say we have some distribution from which we would like sample. So, let’s make a distribution.

We will start with the normal, and we can move into others when we talk about probability distributions and sample statistics in Chapter 5. For this, we will use the distribution of American shad lengths for age-6 females because it approximates a normal distribution. We will calculate the `mean` and `sd` because those are the parameters of the normal distribution.

Start by looking at the size distribution for age 6 females. We use the tidy workflow here with really awful default graphics (more to come in Chapter 4), but we add two arguments to our `subset` call. We want to select only the variable `Length` from `am_shad`, and we want to drop all other information so we can send the output straight to the `hist()` function as a vector.

```
am_shad %>%
  subset(Age == 6 & Sex == "R", select='Length', drop=TRUE) %>%
  hist(main = "")
```



Now, let's calculate the mean and sd of `Length` for age 6 females.

```
# Calculate the mean Length
x_bar <- am_shad %>%
  subset(Age == 6 & Sex == "R", select='Length', drop=TRUE) %>%
  mean

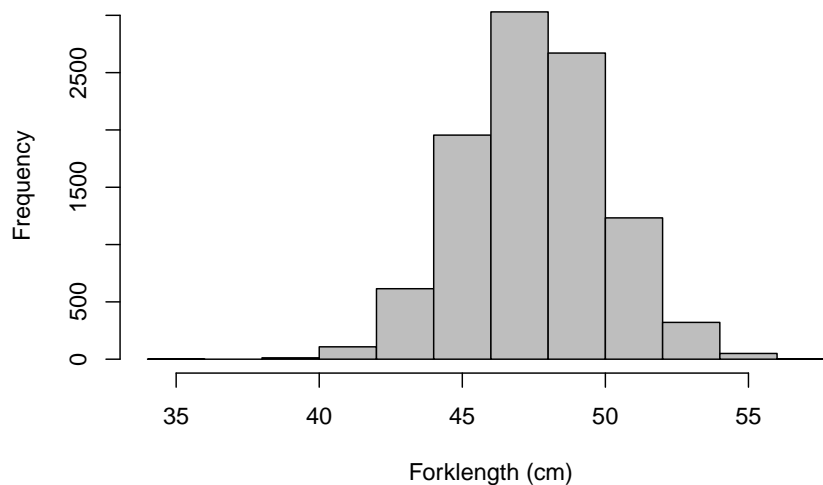
# Calculate standard deviation of Length
sigma <- am_shad %>%
  subset(Age == 6 & Sex == "R", select='Length', drop=TRUE) %>%
  sd
```

Note that we could also use the `filter()` function from the `dplyr` package for this job, and for big data sets it would be a lot faster for un-grouped data.

Now, we can use the mean and standard deviation to randomly sample our normal distribution of lengths.

```
# Take a random sample from a normal distribution
length_sample <- rnorm(n = 10000, mean = x_bar, sd = sigma)

# Plot the sample to see if it is a normal- YAY it is!
hist(length_sample,
      col = "gray",
      main = "",
      xlab = "Forklength (cm)"
    )
```



We've add a couple of new arguments to the histogram call to make it a little less ugly here. In Chapter 4 we are going to ramp it up and play with some plots!

3.7 Next steps

In this chapter, we provided a general overview of our work flow when it comes to reading in data and manipulating them to get useful summaries. In Chapter 4 we will use these processes to help us visualize important trends in these summaries before we begin working with descriptive statistics and sampling distributions in Chapter 5.

Chapter 4

Plotting and graphics

Sweet graphs, right? Want to learn how to make them? Okay, but baby steps here, alright?.

In this Chapter we will walk through plotting in R, both with the base graphic utilities that come with R, and the `ggplot2` package from the `tidyverse` that has taken over the world (er, revolutionized how we write R code). Both of these are actually great to work with for different reasons. The base graphics are built-in, powerful, and give you 100% full control over your graphic environment. The `ggplot2` library (and a million packages that people have written to work with it) takes these to a new level in terms of functionality and 95% of the time it will do exactly what you need. That other 5% of the time it is really great to be able to fall back on the base graphics.

For the examples in this chapter, we'll work with the water quality data contained in `physical.csv`. You will need to download the class data sets that go with this book to play along if you have not already ([click here for instructions from the course website](#)). But, you should have downloaded those to complete the examples in Chapter 2.

We will walk through histograms, scatter plots, line graphs, and boxplots in base graphics and `ggplot2` in this Chapter. Later in the book, we will add examples of how to plot predictions from statistical models alongside raw data using these same tools.

If you installed the `tidyverse` successfully in Chapter 3, then you can load all the packages we'll need by including `library(tidyverse)` at the start of your script:

```
# Chapter 4 Lecture module  
  
# Package load
```

```
library(tidyverse)

# 4.2 Plotting with base R ----
# ...
```

4.1 Plots matter as much as stats

Before we get started:

1. There are few statistical tests that hold intuitive meaning to our readers. The ability to present information in a visually meaningful way to the reader can help to make the interpretation of your science crystal clear without having to worry about whether or not your reader has the faculties to interpret the results of some fancy statistical test that *you* think is cool.
2. Effects and effect sizes are often (if not always) more important than the ability to detect ‘significant’ differences. If you can present clear evidence that some treatment or manipulation confers a biologically meaningful change in a visual way alongside these tests, you can provide a much stronger body of evidence with which to argue your case.
3. There are a few graphical tools that are very useful for basic data exploration, diagnostics, etc., that can make your life a lot easier for data analysis and interpretation. They can also help you decide whether something has gone terribly wrong.

The takeaway here is: *don’t make shitty graphs.*

4.2 Plotting with base R

Let’s look at a few simple types of plots in R. The default graphics in R are not much to look at. But, there are a **ton** of ways to modify these plots, and the user (that’s you!) can build plots from the ground up if needed.

One of the great things about base graphics is that many of the plot types take the same, or similar arguments that are all based on shared graphical parameters.

You can access the help file for these shared graphical parameters by running `?pars` in the console. We will use many of these in the sections that follow.

4.2.1 Histograms

Let's start with the histogram function that we began playing with at the end of Chapter 3.

The `hist()` function plots a histogram but it actually does a whole lot more than just that. Like other plotting utilities, it can take a wide variety of arguments and it actually does some basic data analysis behind the scenes. All of the arguments are optional or have default values with the exception of the data that we want to plot (a numeric variable). This is the case for most plotting functions in the base graphics for R.

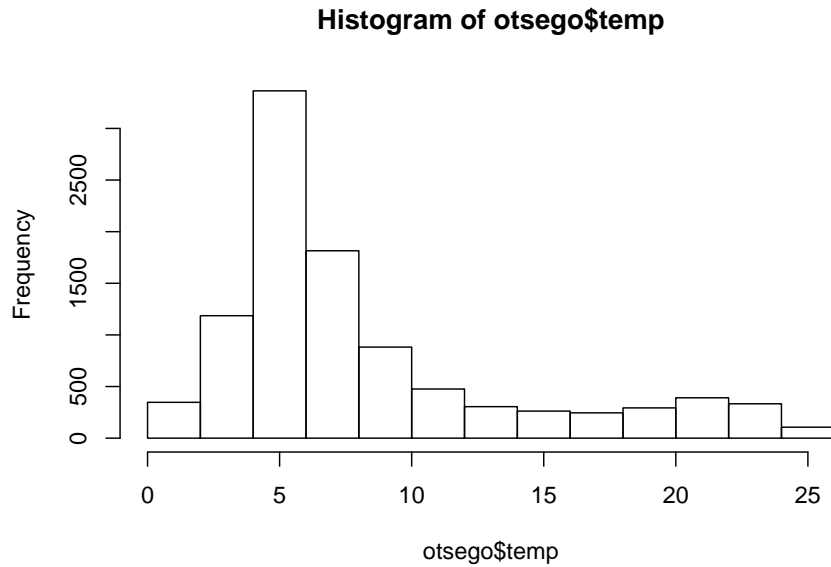
Start by reading in the data contained within the `physical.csv` file from the class data folder. Remember, I am assuming that your code is inside of a folder that also contains your class data folder that you named `data`.

```
# I added stringsAsFactors = FALSE to read in all  
# text strings as `chr`.  
otsego <- read.csv("data/physical.csv", stringsAsFactors = FALSE)
```

These are data collected each year from Otsego Lake by students, staff, and faculty at the SUNY Oneonta Biological Field Station in Cooperstown, NY, USA. The data set includes temperature (°C), pH, dissolved oxygen, and specific conductance measurements from a period of about 40 years. There are all kinds of cool spatial and seasonal patterns in the data that we can look at. We will use `temperature` for the examples that follow.

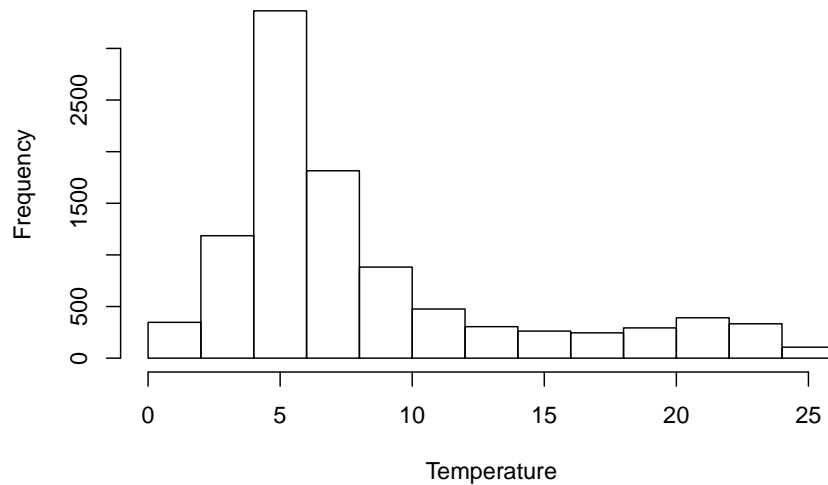
Make a histogram of temperature across all depths and dates just to see what we are working with here:

```
hist(otsego$temp)
```



The default histogram in base graphics leaves much to be desired. Thankfully, it is easy to modify the look of these figures. For example, we can add labels to the x and y-axis using `xlab` and `ylab`, and we can give the histogram a meaningful title by adding `main = ...` to the `hist()` call or remove it completely by saying `main = ""`.

```
hist(otsego$temp, xlab = "Temperature", ylab = "Frequency", main="")
```

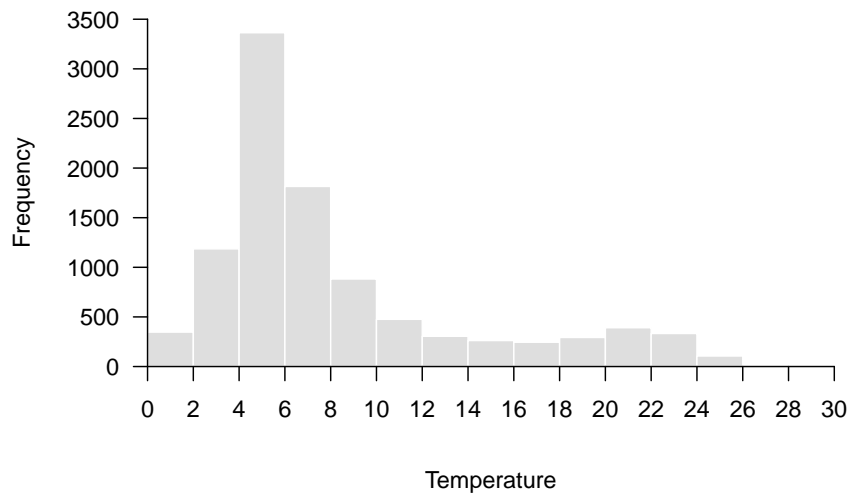
We can make the axes cross at zero if we are concerned about that. We need to do this by specifying `yaxt = "n"`, `xaxt = "n"` in the `hist()` call and then follow up by telling R exactly where to start each of the axes. In this example, I also add some changes to the color of the bars (`col`) and the color of the borders (`border`). Finally, I fix the x- and y-axis scales so I know where they'll start.

```
# Make the histogram
hist(otsego$temp,
     xlab = "Temperature",
     ylab = "Frequency",
     main = "",
     xaxt = "n",
     yaxt = "n",
     col = "gray87",
     border = "white",
     xlim = c(0, 30),
     ylim = c(0, 3500)
)

# Add an x-axis going from zero to thirty degrees
# in increments of 2 degrees and start it at zero
axis(side = 1, at = seq(from = 0, to = 30, by = 2), pos = 0)

# Add a rotated y-axis with default scale and
# start it at zero
```

```
axis(side = 2, las = 2, pos = 0)
```



Colors!!!

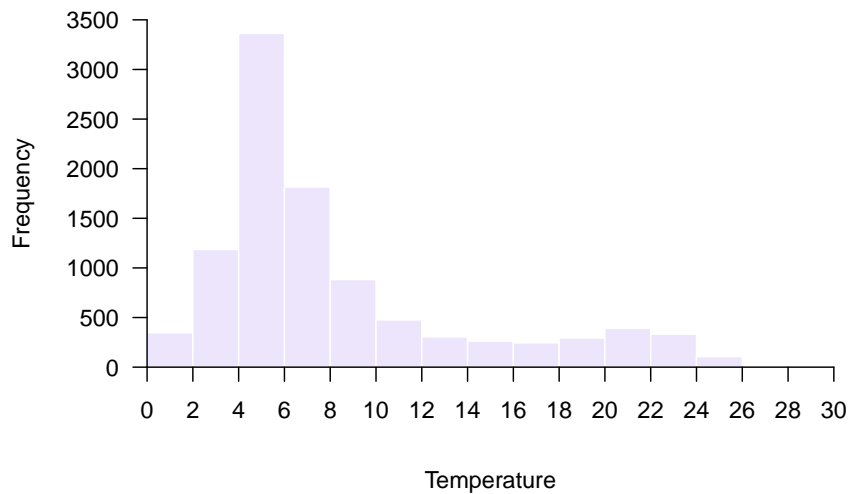
If `gray87` is not your style (whatevs), there are another 656 pre-named colors in R. You can see their names by running `colors()` in the console like this:

```
colors()
```

If you are a little more adventurous, you might try the `rgb()` color specification or hex values. I really like the `rgb()` specification because you can include an `alpha` channel to make your colors transparent (oooooh!). For example, if I change my code above to use the following:

```
col = rgb(red = 0.90, green = 0, blue = 0.30, alpha = 0.10)
```

I get a transparent, purple histogram.



So purply.

There are tons of great blogs and eBooks with whole chapters devoted to colors and color palletes in R. There are even whole packages we'll work with dedicated to colors. By all means, check them out! We will work with a few as we continue to increase complexity.

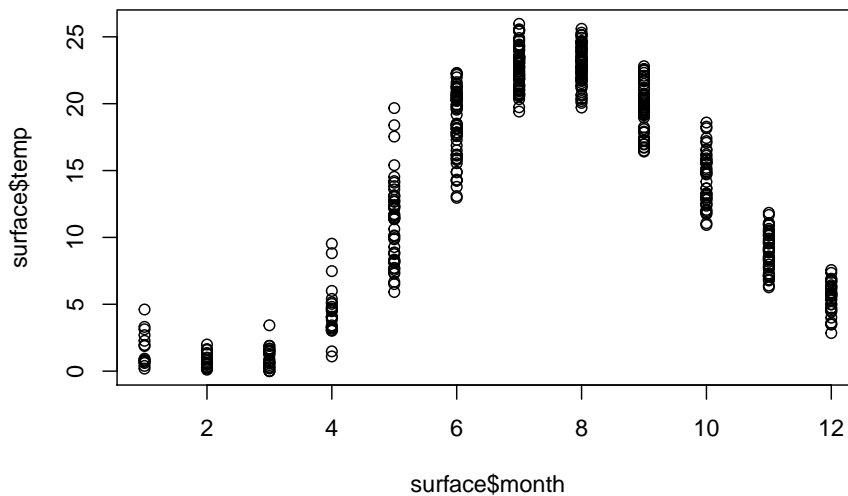
4.2.2 Scatterplots

Scatter plots are a great starting point for doing exploratory data analysis or for displaying raw data along with summary graphics. They are also the default behavior for the `plot()` function for continuous variables in base R.

Let's demonstrate by plotting surface temperature (`depth = 0.1` m) by month across years. We'll use the data management skills we picked up in Chapter 3 to filter the data first.

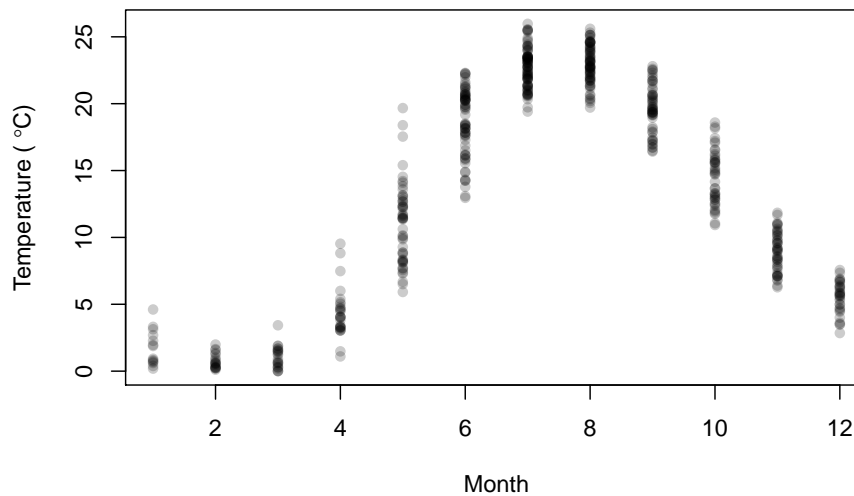
```
# Filter to get July surface temperatures
surface <- otsego %>% filter(depth == 0.1)

# Default scatter plot
plot(x = surface$month, y = surface$temp)
```



As with the `hist()` function, the default here is underwhelming. We can use many of the same arguments that we specified in `hist()` to dress this up a bit. This time, we will specify a plotting character `pch` that corresponds to a filled circle. Then, we tell R to give it an `rgb()` background (`bg`) with no color for lines that go around each point. That way the data points are darker where there is overlap between them. Finally, we use `expression()` to include the degree symbol in the y-axis label.

```
# Better scatter plot
plot(x = surface$month,
     y = surface$temp,
     pch = 21,
     bg = rgb(0, 0, 0, 0.2),
     col = NA,
     xlab = "Month",
     ylab = expression(paste("Temperature ( ", degree, "C)"))
)
```



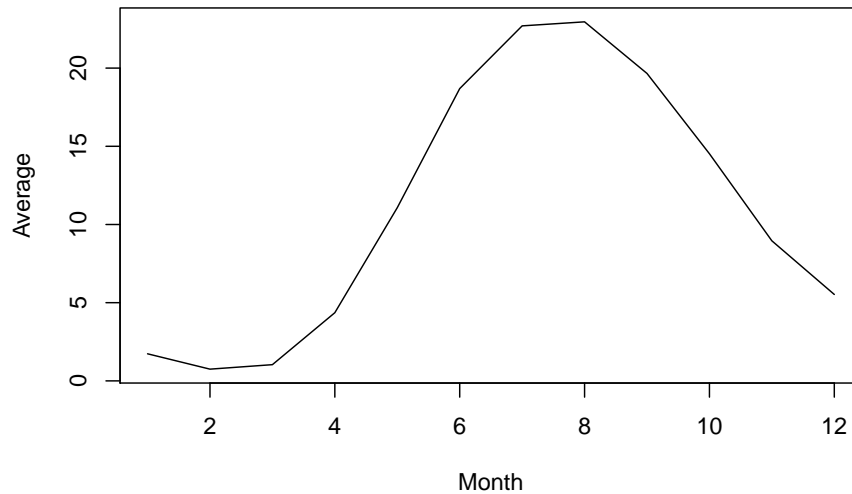
This is a lot more informative because it shows us where most of the observations fall within a given month, and how much variability there is. But, it would be nice to have some summary.

4.2.3 Lines

We can plot lines in a few different ways in the base graphics of R. We can create stand-alone line graphs with data in R pretty easily with the `plot()` function we used for scatter plots in the preceding section.

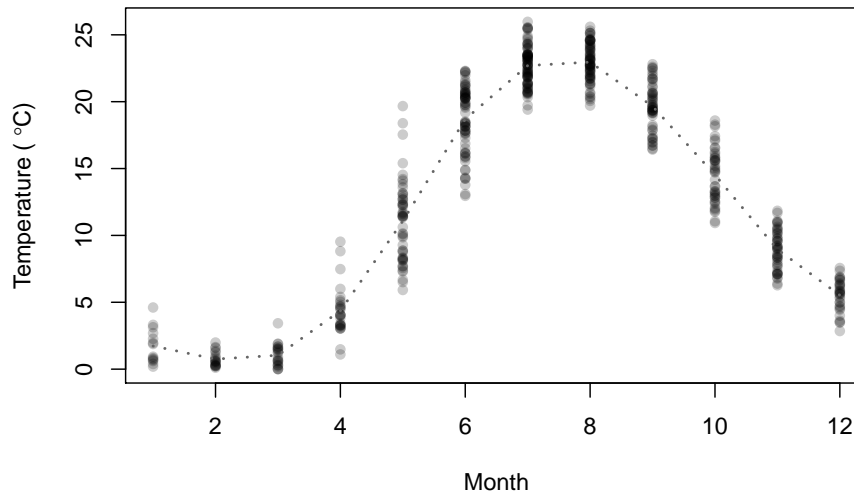
For example, let's say that we want to just plot average surface temperature in each month as a line graph. We can summarize the data quickly and then plot those:

```
mids <- surface %>%  
  group_by(month) %>%  
  summarize(avg = mean(temp))  
  
plot(mids$month, mids$avg, type = "l", xlab = "Month", ylab = "Average")
```



We could even add these to the scatter plot of our raw data using the `lines()` function. Play around with `lty` and `lwd` to see if you can figure out what they do. If you get stuck, don't forget to Google it! (Worst Stats Text ever.)

```
# Same scatter plot
plot(x = surface$month,
     y = surface$temp,
     pch = 21,
     bg = rgb(0, 0, 0, 0.2),
     col = NA,
     xlab = "Month",
     ylab = expression(paste("Temperature ( ", degree, "C)"))
)
# Add a thick, dotted line that is gray (this is a gray40 job)
lines(mids$month, mids$avg, lty = 3, lwd = 2, col = "gray40")
```



We could also add the means to the main plot with `points()` and choose a different size or color than the raw data. We'll look at these options and more as we step up complexity.

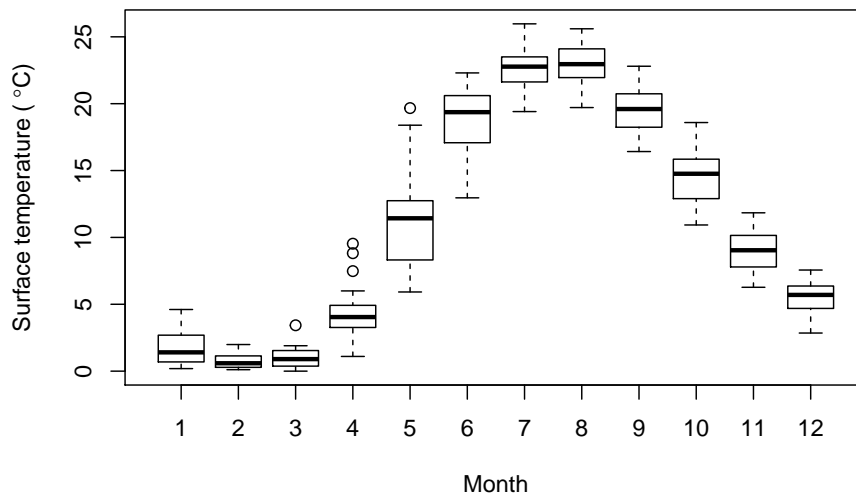
For raw data like these, though, we are better off using a box plot to show those types of summaries.

4.2.4 Boxplots

The basic box plot is straightforward to create, but can be a real pain to modify because the syntax is slightly different from the other plotting functions we've worked with so far.

Let's try to summarize surface temperature by month using a box plot to see how these work in the base R graphics. Notice that we are specifying the variables as a formula here, and explicitly telling R what the data set is:

```
boxplot(temp ~ month, data = surface,
        xlab = "Month",
        ylab = expression(paste("Surface temperature ( ", degree, "C)")))
```



Wow, that was waaaaay to easy! Have you ever tried to make one of those in Excel? Forget about it. It would take you half a day, and then when you realized you forgot ten data points you would have to do it all again.

But, it is still much ugly. Maybe there is a way we can change that?

Of course there is!

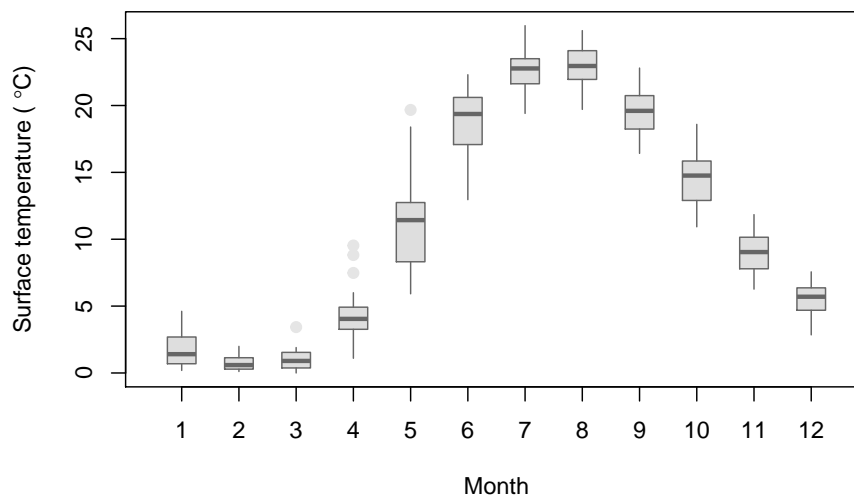
Let's add a little color and tweak some options. For a full set of optional arguments you can change, run `?bxp` in the console. (Ooh, that one is sneaky: `bxp()` is the function inside of `boxplot()` that actually draws the plots).

Options are named consistently by the part of the plot. For example, `boxwex`, `boxcol` and `boxfill` all control the look of the box. Likewise, the options `boxcol`, `whiskcol` and `staplecol` control the colors of the box, whiskers, and staples, respectively. Nifty, right? Play with the settings below to see what each one does. Then, go explore some more options. It is the easiest way to learn when you are learning from The Worst Stats Text ever.

```
boxplot(temp~month,
  data = surface,
  xlab = "Month",
  ylab = expression(paste("Surface temperature ( ", degree, "C)")),
  border = "gray40",
  boxwex = 0.50, boxcol = "gray40", boxfill = "gray87",
  whisklty = 1, whisklwd=1, whiskcol = "gray40",
  staplewex = 0, staplecol = NA,
```

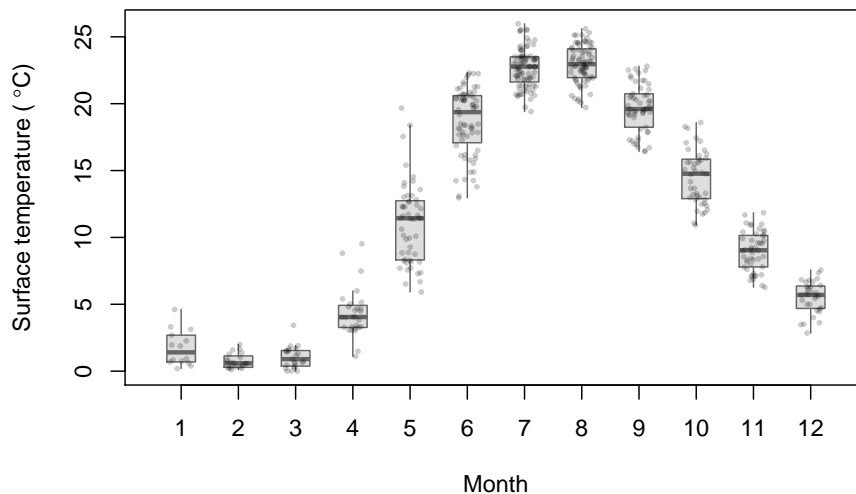


```
outpch = 21, outbg = "gray90", outcol = "gray90"
)
```



Finally, we can combine this with a scatter plot to **jitter** our raw data over the top of the boxes in each month:

```
boxplot(temp~month,
  data = surface,
  xlab = "Month",
  ylab = expression(paste("Surface temperature ( ", degree, "C)")),
  border = "gray40",
  boxwex = 0.50, boxcol = "gray40", boxfill = "gray87",
  whisklty = 1, whisklwd=1, whiskcol = "gray40",
  staplewex = 0, staplecol = NA,
  outpch = 21, outbg = NA, outcol = NA
)
points(jitter(surface$month), surface$temp, cex=.4, pch=19, col=rgb(0,0,0,0.2))
```



That is actually starting to look pretty snazzy! We'll continue to work to improve our base graphics as we move forward. For now, let's have a look at how to do these things in `ggplot2` next.

4.3 Plotting with `ggplot2`

Plotting with `ggplot2` and the dozens of packages that use it is a bit different than plotting with base graphics in R. Part of the reason for this is that it uses a work flow that is similar to the data manipulation we have looked at so far. In general, you could think of this as creating a canvas, applying some routine aesthetics based on the data structure (e.g. grouping), and then adding layers on to the canvas like we did with base graphics.

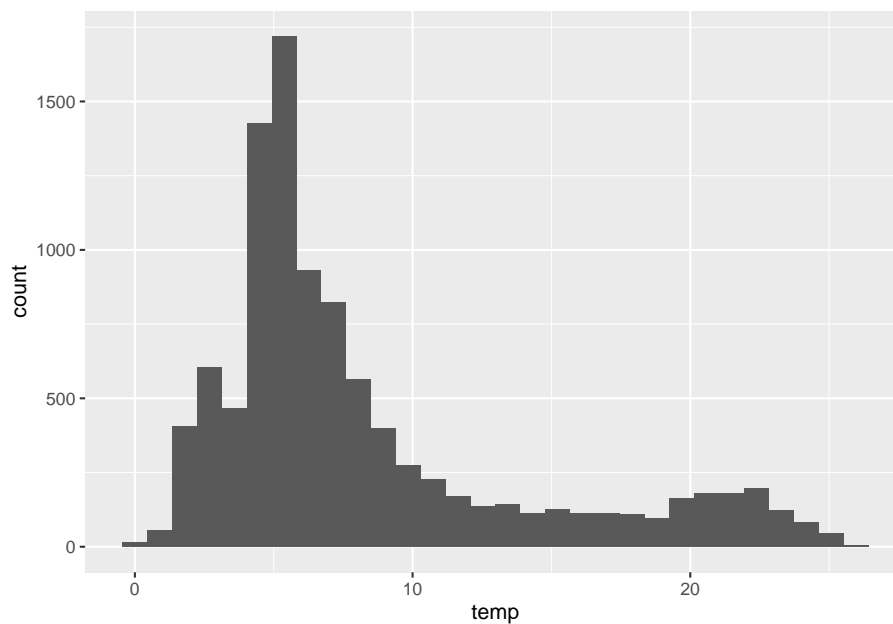
It takes a little getting used to, but you'll see how powerful it can be for multi-faceted plots when we get to later chapters. We'll walk through the same plots that we did in base graphics, but this time we'll use the `ggplot()` function and layer on the pieces.

4.3.1 Histograms

The default histogram is easy to create with `ggplot()` and a geometry layer. We start with the `ggplot` call, and then add the histogram geometry, `geom_histogram()`, like this. I usually save the plot to an object with an

arbitrary name I don't use for anything, like `p` or `s` or `v`, and then print it explicitly.

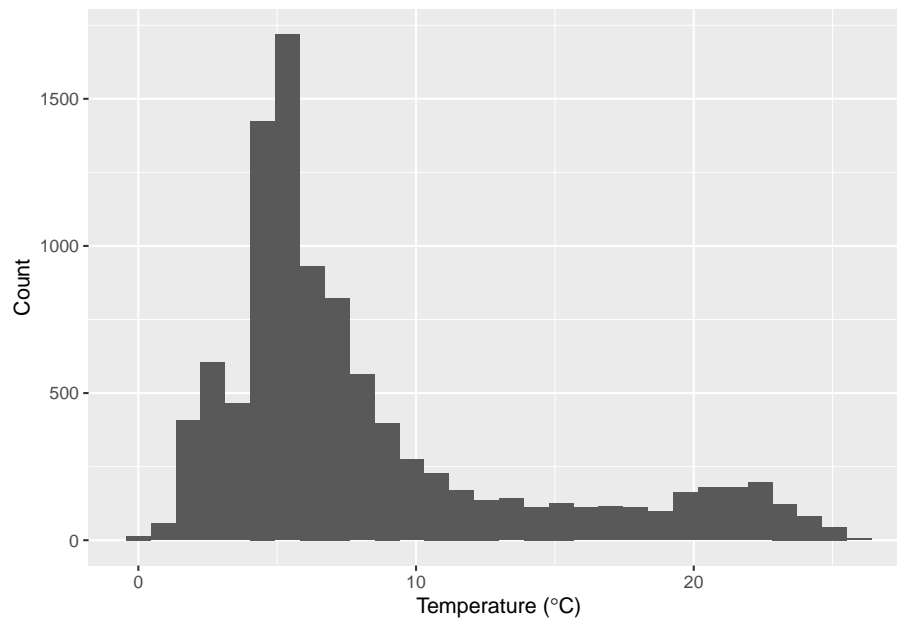
```
# Histogram of water temperature across  
# all dates and depths  
p <- ggplot(otsego, aes(x=temp)) + geom_histogram(bins=30)  
  
print(p)
```



Right away this looks a lot prettier than the default histogram that we produced with base graphics. Of course, we can customize just like we did before.

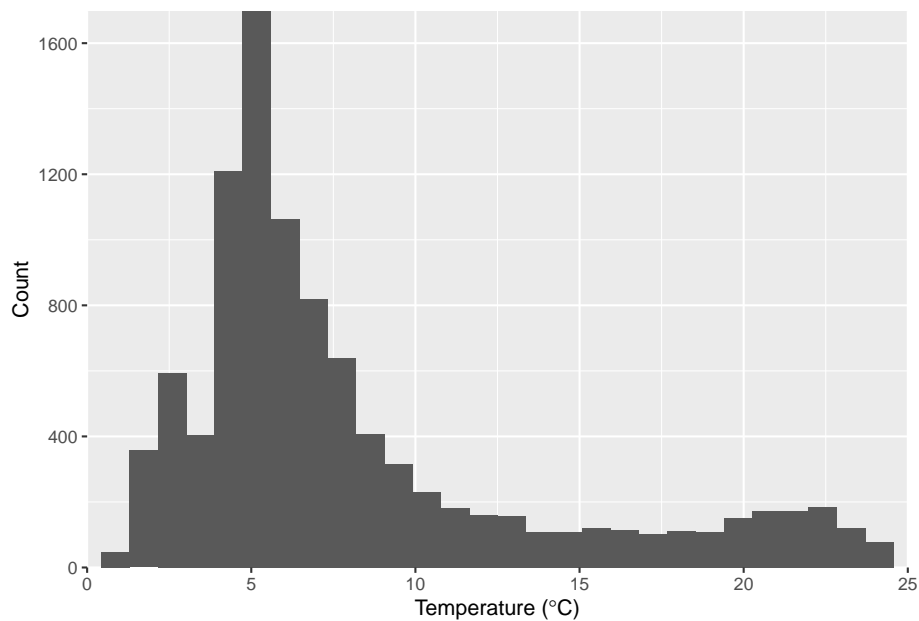
Let's add labels for the x and y-axis next:

```
# Histogram of water temperature across  
# all dates and depths  
p <- ggplot(otsego, aes(x=temp)) +  
  geom_histogram(bins=30) +  
  xlab(expression(paste("Temperature (", degree, "C)"))) +  
  ylab("Count")  
  
print(p)
```



We can also scale the `x-axis` and the `y-axis` like we did in the base graphics example.

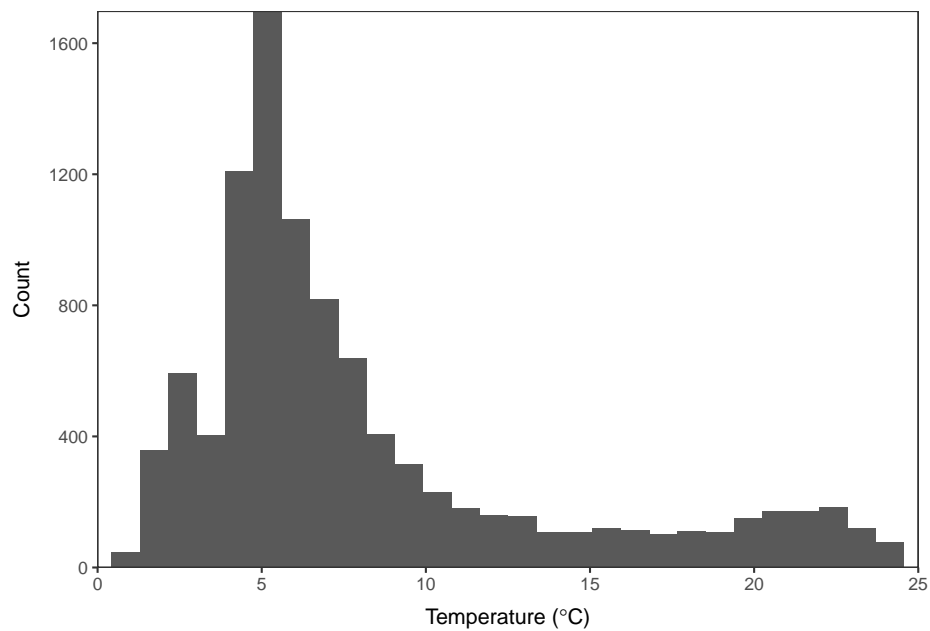
```
# Histogram of water temperature across  
# all dates and depths  
p <- ggplot(otsego, aes(x=temp)) +  
  geom_histogram(bins=30) +  
  xlab(expression(paste("Temperature ", degree, "°C"))) +  
  ylab("Count") +  
  scale_x_continuous(limits=c(0, 25), expand = c(0, 0)) +  
  scale_y_continuous(expand = c(0, 0))  
print(p)
```



We can modify each of other layers individually, all at once using preset ggplot themes or by modifying a pre-defined theme.

Let's have a look at how a theme changes the appearance. I am going to add `theme_bw()` here, but check out the others linked above. I also add a few adjust the position of the x- and y-axis labels and removed the panel grid in the `theme()` function after applying a theme.

```
# Histogram of water temperature across
# all dates and depths
p <- ggplot(otsego, aes(x=temp)) +
  geom_histogram(bins=30) +
  scale_x_continuous(limits=c(0, 25), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab(expression(paste("Temperature (", degree, "C)"))) +
  ylab("Count") +
  theme_bw() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```



Spend some time practicing this and changing options to see what you can come up with. Be sure to check out the descriptions of the options you can pass to theme by running `?theme` to get the help file.

4.3.2 Scatter plots

Now that we have a basic feel for the `ggplot2` work flow, changing plot types is really easy because all of the parts of our plots work together in the same way.

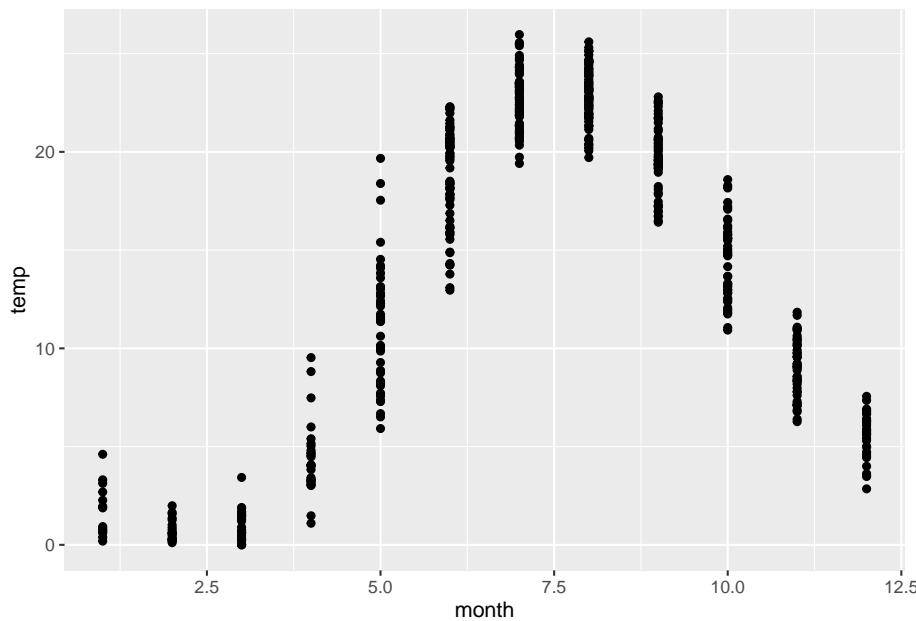
As a reminder, we previously built scatter plots of surface temperature in Otsego Lake, NY by month using base graphics.

Go ahead and subset the data again:

```
surface <- otsego %>% filter(depth == 0.10)
```

Now, we can make the default scatterplot:

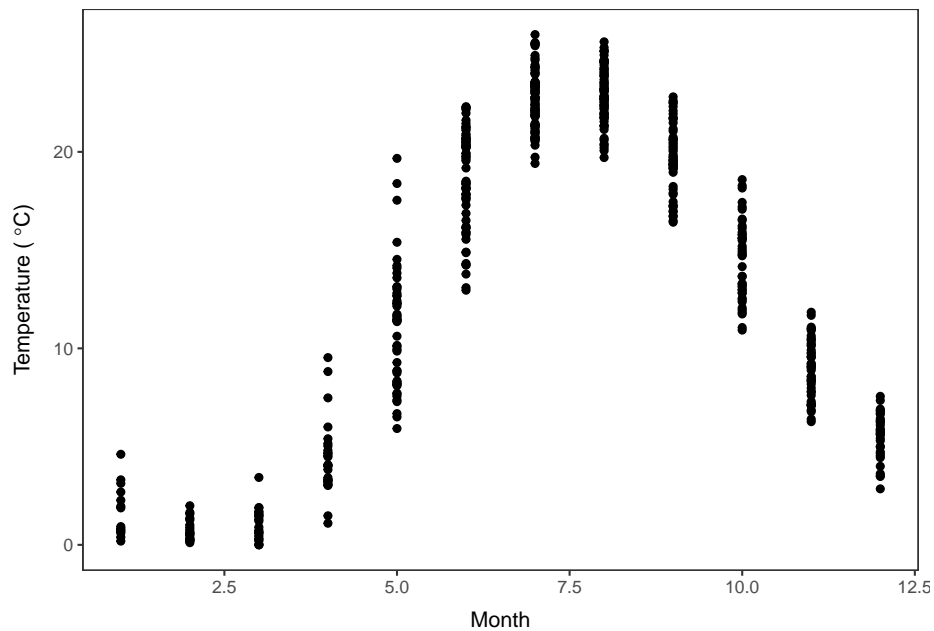
```
s <- ggplot(surface, aes(x = month, y = temp)) +  
  geom_point()  
print(s)
```



At a glance, this already looks a lot nicer than the default scatterplots from base graphics, but we still have a lot of work to do. Plus, we get ggplots own odd behaviors when it comes to the x-axis scale and titles. So, let's get to work!

First, we'll replace the default axis titles and add the `theme_bw()` that we used above, with the same modifications to axis positions and grid lines.

```
s <- ggplot(surface, aes(x = month, y = temp)) +
  geom_point() +
  xlab("Month") +
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank())
print(s)
```



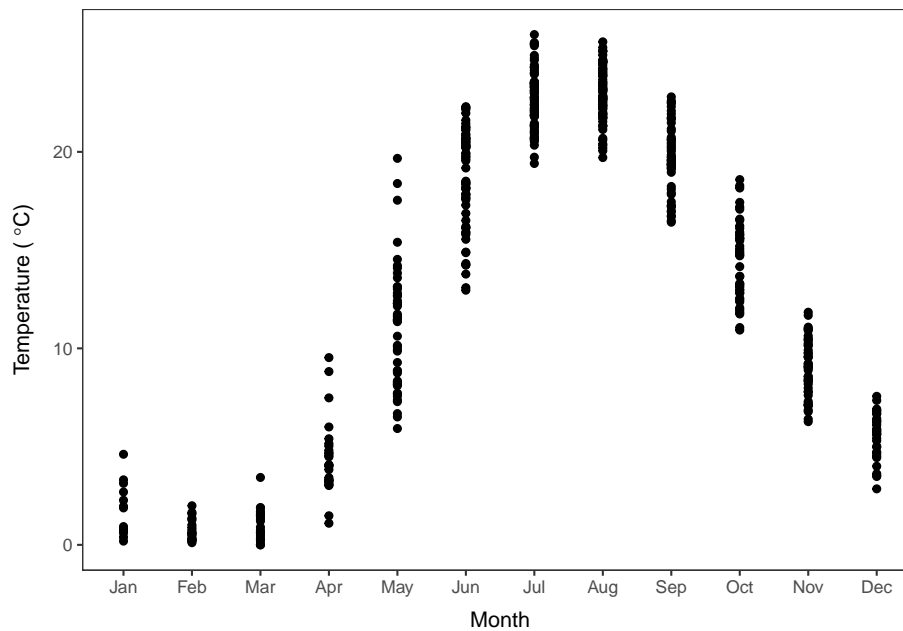
Okay, now we need to fix that pesky x-axis scale to use whole months or text labels.

To fix the axis scales, we've actually got to do a little bit of work this time. In this case the easiest thing to do is probably to make a categorical variable out of the column `month`, which is an integer. We can do this using some fancy indexing with the built-in object that contains month abbreviations, `month.abb` in base R.

```
surface$c_month <- factor(month.abb[surface$month], levels=month.abb)
```

Whoa, that was a heavy lift (sarcasm). Let's see how that changes the appearance of our plot:

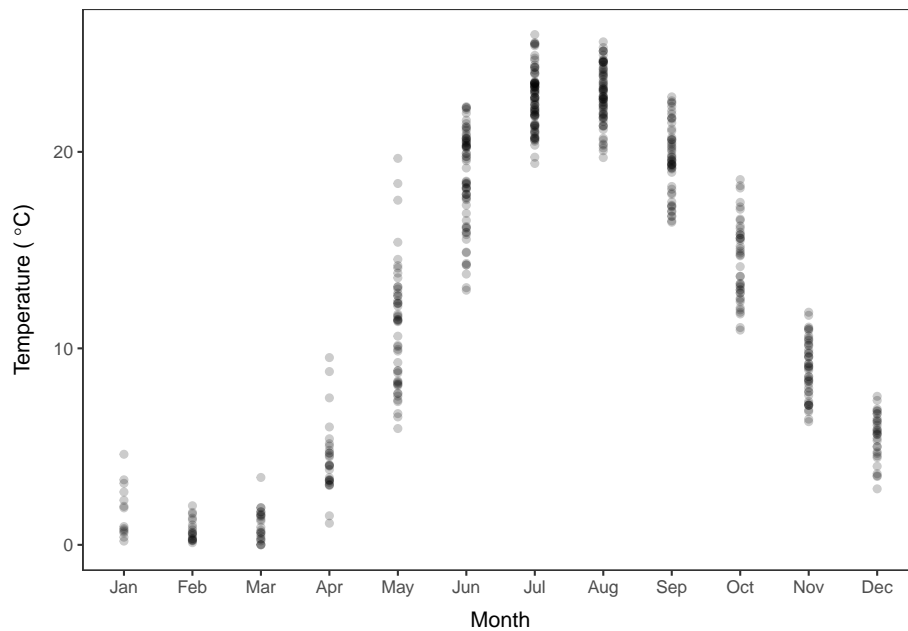
```
s <- ggplot(surface, aes(x = c_month, y = temp)) +
  geom_point() +
  xlab("Month") +
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank())
print(s)
```

This is starting to look really nice.

Finally, we just add a little transparency to the points by specifying `alpha = 0.2` inside of `geom_point()` and we are good to go!

```
s <- ggplot(surface, aes(x = c_month, y = temp)) +  
  geom_point(alpha = 0.2) +  
  xlab("Month") +  
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +  
  theme_bw() +  
  theme(axis.title.x = element_text(vjust = -1),  
        axis.title.y = element_text(vjust = 3),  
        panel.grid = element_blank())  
print(s)
```



Looks just like the one we made with base graphics!

4.3.3 Lines

Most of the time we plot line graphs, whether in base graphics or using `ggplot2`, we are going to be adding them to existing plots. This was really straightforward in base graphics. It is only slightly more complicated in `ggplot2`.

We'll start with the default line graph, and then add it to the scatter plot from the previous section.

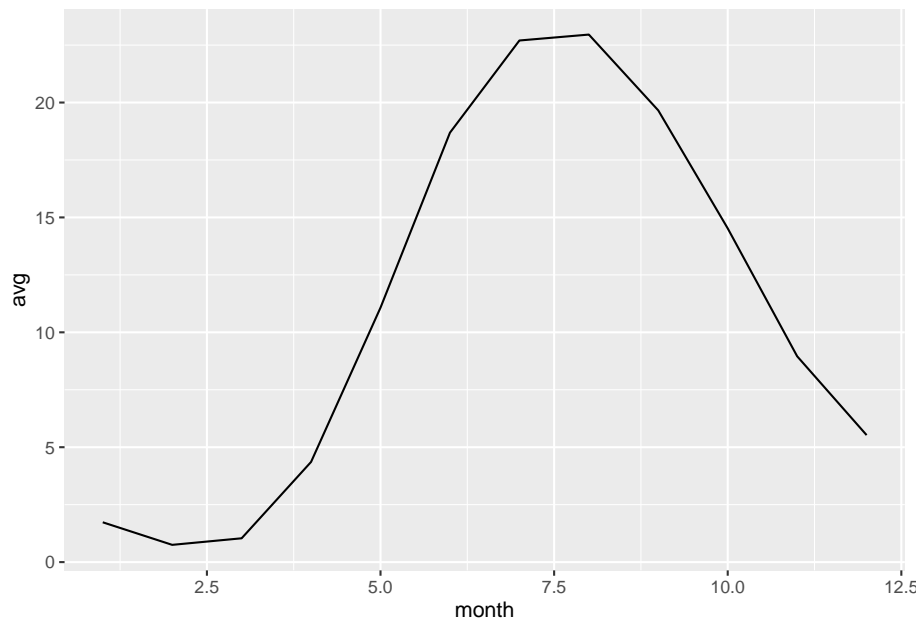
Let's calculate monthly means of surface temperature in Otsego Lake again:

```
mids <- surface %>%
  group_by(month) %>%
  summarize(avg = mean(temp))
```

Now plot it with `ggplot()`:

```
lp <- ggplot(mids, aes(x = month, y = avg)) +
  geom_line()

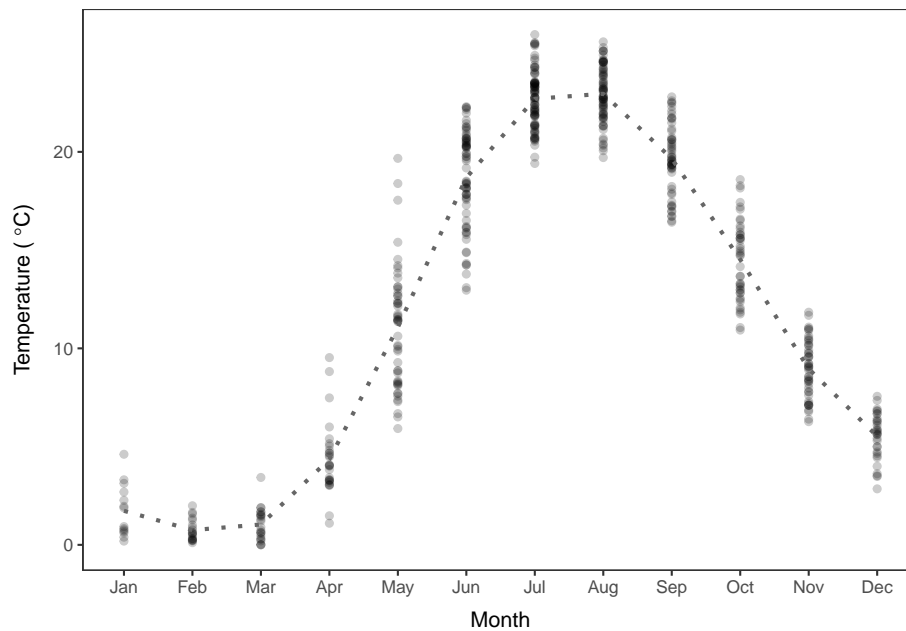
print(lp)
```



There you have it!

Now, we just need to add this to our scatterplot that we made previously. To do this, we have to insert `geom_line()` in the code, but we must specify

```
s <- ggplot(data = surface, mapping = aes(x = c_month, y = temp)) +
  geom_point(alpha = 0.20) +
  geom_line(mapping = aes(x = month, y = avg),
            data = mids,
            color = 'gray40',
            lty = 3,
            lwd = 1) +
  xlab("Month") +
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank())
print(s)
```



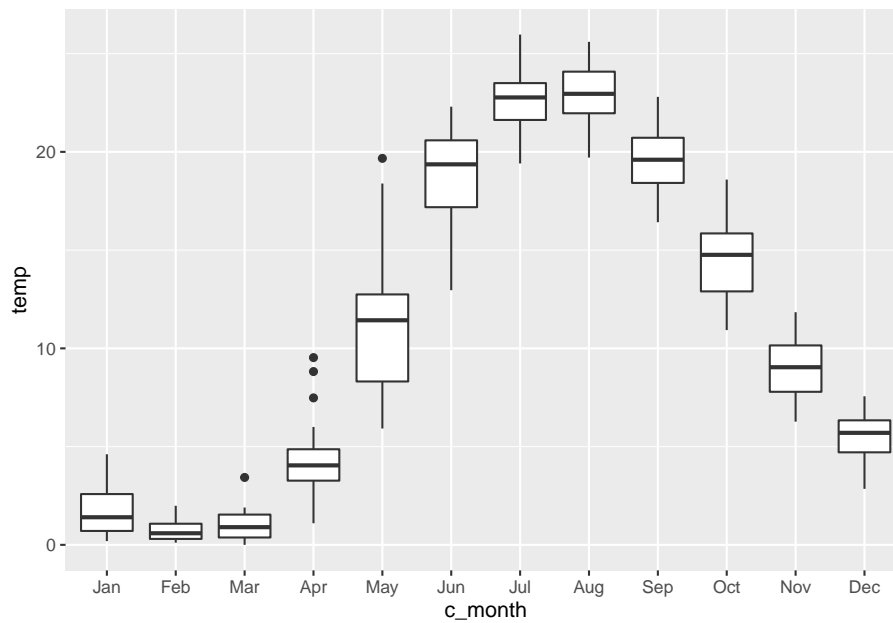
We will continue to use this approach throughout the book to plot raw data and model predictions. So, if it is giving you trouble now, spend some extra time with it.

4.3.4 Boxplots and

To wrap up our tour of plotting examples in `ggplot2`, we will reproduce (more or less) the box plots we made in base graphics.

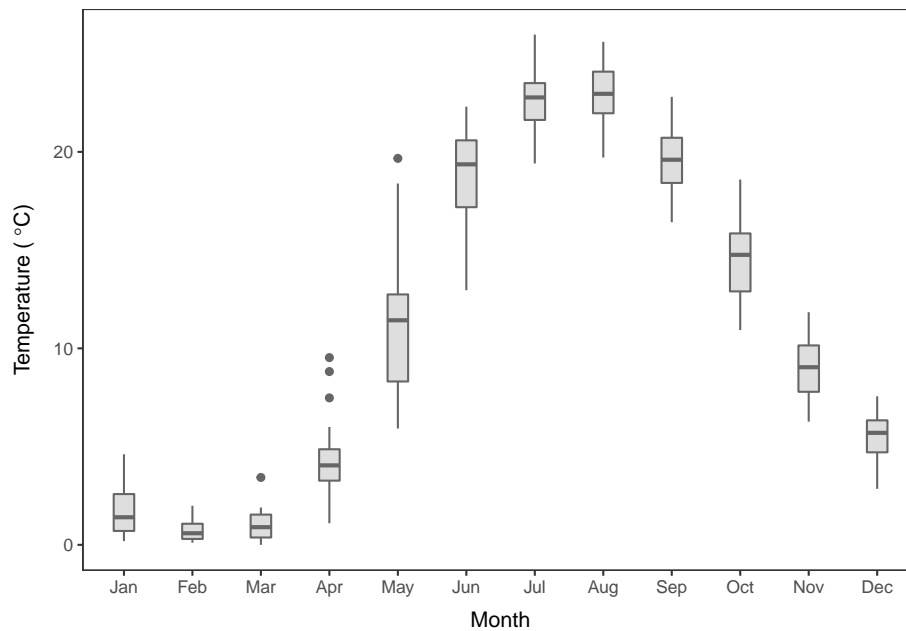
Make the default box plot of surface water temperatures in Otsego Lake, NY. Notice that we use the `c_month` variable that we made previously in the `surface` data so R knows these are groups.

```
bp <- ggplot(surface, aes(x = c_month, y = temp)) + geom_boxplot()
print(bp)
```



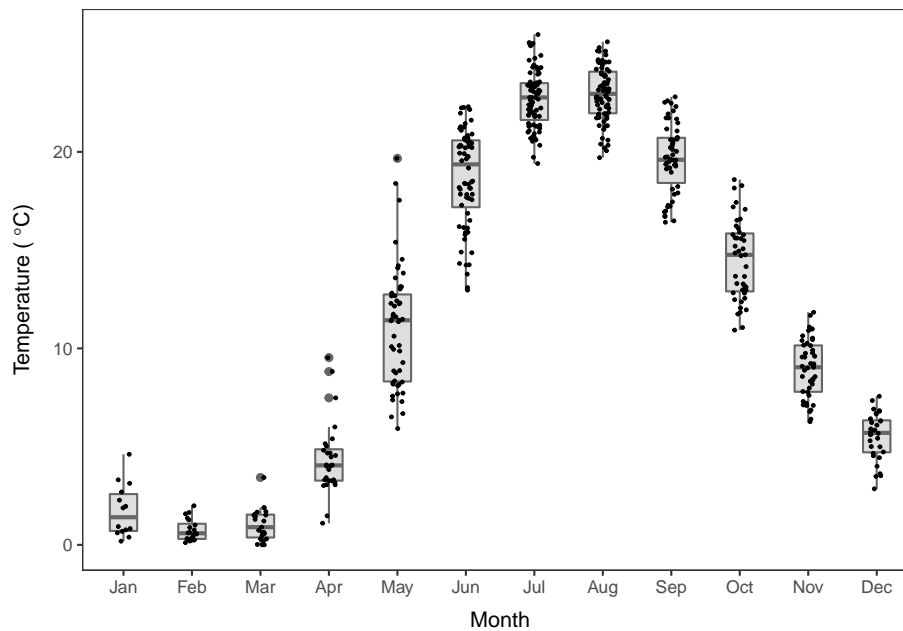
If we add changes we made to previous plots here, then we can get a cleaner look:

```
bp <- ggplot(surface, aes(x = c_month, y = temp)) +  
  geom_boxplot(color = 'gray40', fill = 'gray87', width = 0.3) +  
  xlab("Month") +  
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +  
  theme_bw() +  
  theme(axis.title.x = element_text(vjust = -1),  
        axis.title.y = element_text(vjust = 3),  
        panel.grid = element_blank())  
  
print(bp)
```



And, of course, we can add our “jittered”, raw data points over the top to show the spread.

```
bp <- ggplot(surface, aes(x = c_month, y = temp)) +
  geom_boxplot(color = 'gray40', fill = 'gray87', width = 0.4) +
  geom_jitter(size = .5, width = 0.1) +
  xlab("Month") +
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank())
print(bp)
```



4.4 Next steps

Hopefully this chapter provided you with a basic overview of plotting in R. If you struggled with these exercises, practice them again, or check out some additional online resources. In the coming chapters, we will continue to add functionality and complexity to how we use these kinds of plots. We'll look at how to compare two groups within a single plot in Chapter 6 when we dive into sampling distributions. And, eventually we will learn how to plot predictions from our statistical model against observed data from Chapter 8 onward.

Chapter 5

Sampling distributions in R

If we can describe the shape of a probability distribution for a random variable like temperature we can make predictions about the world. Sinister? Maybe. These are temperatures that I simulated from the Hudson River using historical data to estimate parameters of a multivariate normal distribution (muwahaha-haha).

In this Chapter, we'll talk about probability and probability distributions as a backdrop for the models that we will be working with during the next several chapters. When we describe these from data we have collected, we call them **sampling distributions**. Probability theory is central to statistical techniques, so it will be important for you to have a pretty firm understanding of this to grab hold of big ideas later on. For now, play along and try to understand how they work. We'll swing back later for a refresher.

In order to complete this Chapter, you will need to have the `ggplot2`, `MASS`, and `Rlab` packages loaded. The only one you should need to install is the `Rlab` package because `MASS` is installed when you install R, and we already installed `ggplot2` with the `tidyverse` in Chapter3.

I'm going to load these now. In general, it is good practice to put these at the top of the script so we know they are needed.

```
library(ggplot2)
library(MASS)
library(Rlab)
```

None of the class data are required to complete this chapter.

5.1 What are sampling distributions?

When we talk about sampling distributions, we are talking about the probability that a variable we can measure (e.g. temperature) takes on some value. In most cases, there is a higher probability that the variable will take on certain values than others. That probability may be governed by any number of processes and thus may assume a number of different shapes with respect to the likelihood of any given value of our variable. The differences in the shapes that we assume, and the mathematical parameters that we use to describe those shapes are called “probability distributions”. And, when they are estimated from data, they are sampling distributions.

There was a time when biologists were largely restricted to using models that relied heavily on the assumption that the things we measured, and their errors, followed “normal” distributions, which you have probably heard of or seen in a scientific paper. This was because of how computationally intensive other methods were. This often led to the use of strictly parametric tools like ANOVA and t-tests, or the use of strictly non-parametric tools like frequency analyses and rank-order methods. While these are still useful techniques in our tool-boxes, that time has passed, and now we have access to a wide range of tools that allow us to extend simple parametric and non-parametric tools to relax or change distributional assumptions. We will discuss these throughout the book, but we need to look at the underlying distributions that govern our decisions about which of these tools to use. So, this week we’ll look at a few probability distributions that correspond to sampling distributions we frequently encounter in biology. To wrap-up, we will use this new information to talk about how we calculate descriptive statistics such as means and standard deviations from samples.

5.2 Probability distributions in R

R has a number of built-in distribution types, and there are random-number generators associated with most or all of these that will allow us to take random samples from a distribution (like picking numbers out of a hat!). This is useful for data simulation, but is also helpful for us to learn about probability distributions and how their parameters affect the shape, spread, scale, location, etc. of those distributions. We will briefly discuss concepts like skew because of how they can help us think about the assumptions that we are making (or breaking!) in the models that we use.

For this class, we will focus on one major family of distributions and then zero in on a few distributions within this family that you are guaranteed to encounter throughout your career.

5.3 Exponential family

Most or all of the distributions we will use for this class come from the **exponential family** of distributions.

The exponential family is very flexible, and whether you know it or not, it includes most of the distributions one might use to describe every day phenomena. It includes most of the probability distributions with which you are familiar, and many more. Just ask this *very* reliable Wikipedia entry. Oh, let's face it, you were going there anyway, I just cut out the Google step.

Take a look at the table at the bottom of this Wikipedia page just to get an idea of how many distributions are included within the exponential! Holy cow! We're not going to look at all of these in this class- I just want you to be aware that this is a **huge** family of specific distributions.

Distributions that we'll focus on in this chapter:

1. **Continuous distributions** Normal (Gaussian) Lognormal Beta Uniform
2. **Discrete distributions** Bernouli Binomial Multinomial Poisson Negative binomial

5.4 Continuous distributions

The normal distribution

This is one distribution with which most of you have at least some nodding acquaintance. It is the classic "bell curve" that college students once dreaded in upper-level courses. I don't know if it's a thing anymore. Go Google it.

The **normal distribution** is defined by two parameters:

1. The mean (μ)
2. The variance (σ^2)

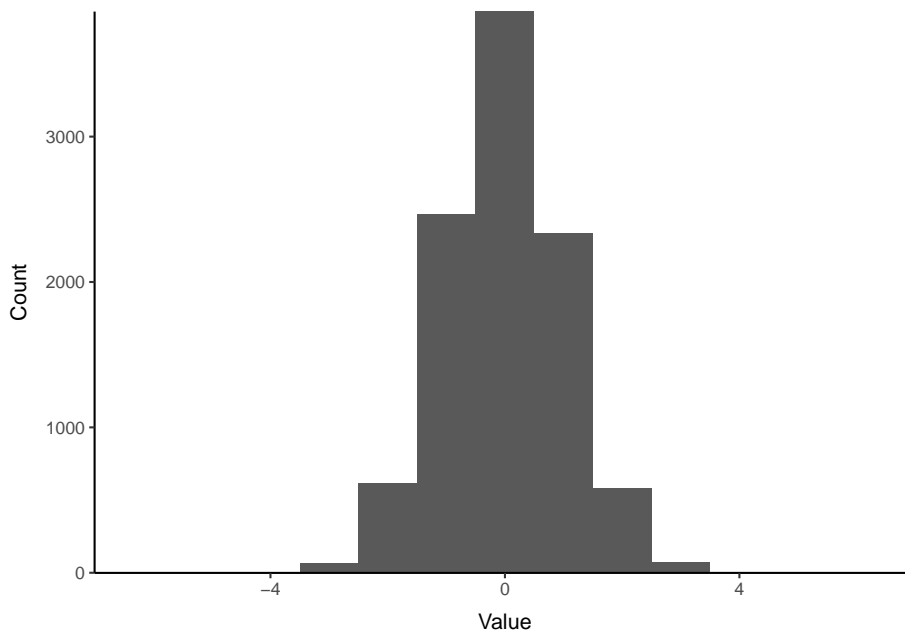
Let's take a look at what the normal distribution looks like. We'll start with a special one called the standard normal (or z) distribution. The standard normal is a normal distribution with a mean of zero and a variance of 1. This one is really cool because the standard deviation (σ) is the square-root of the variance, and in this special case $\sqrt{1} = 1$, so the variance and standard deviation are equal! And because of this property, and other normal distribution can be converted to a standard normal using z -standardization, which we'll talk about later. How exciting is that?

First, take a sample from a normal distribution:

```
samp <- rnorm(n = 10000, mean = 0, sd = 1)
```

Now, plot a histogram using the sick new skills you got in Chapter 4

```
p <- ggplot() +  
  geom_histogram(aes(samp), binwidth = 1) +  
  scale_x_continuous(limits=c(-7,7), expand = c(0, 0)) +  
  scale_y_continuous(expand = c(0, 1)) +  
  xlab("Value") +  
  ylab("Count") +  
  theme_classic() +  
  theme(  
    axis.title.x = element_text(vjust = -1),  
    axis.title.y = element_text(vjust = 3),  
    panel.grid = element_blank()  
  )  
print(p)
```



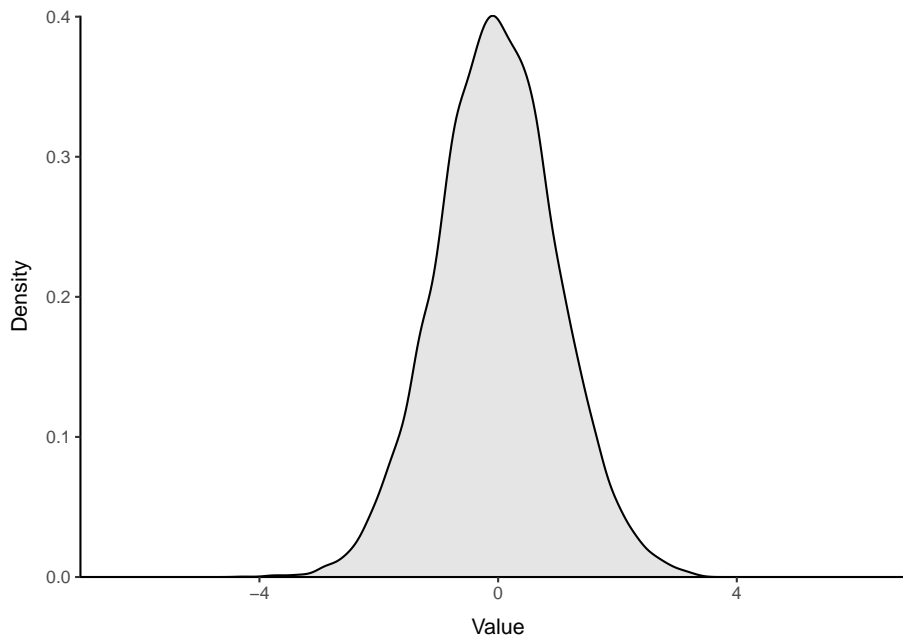
Pretty!

Because this sample is from a **continuous** distribution, we might actually wish to represent this distribution with a probability density function. You can think of this as R calculating the relative probability of an given value. It implies a continuous surface, rather than the discrete bins like the histogram. In reality it doesn't matter because at best we chop continuous distributions into tiny little

bins when we do things like integrals, and R bases `binwidth` in histograms off a density function anyway (aaaaah!).

Density plots are a new one for us, so let's try them out. If you scroll back and forth, you'll notice that the code below is basically identical to the histogram above except for labels and scales. We just replaced the histogram geometry (`geom_histogram`) with a density-based geometry (`geom_density`). Here, we use `fill = 1` to trick R into filling the area under the line because we have no grouping variables. By default, this is interpreted as 'black', so we add an alpha channel for transparency.

```
p <- ggplot() +  
  geom_density(aes(samp), alpha = .1, fill = 1) +  
  scale_x_continuous(limits = c(-7,7), expand = c(0, 0)) +  
  scale_y_continuous(expand = c(0, 0)) +  
  xlab("Value") +  
  ylab("Density") +  
  theme_classic() +  
  theme(  
    axis.title.x = element_text(vjust = -1),  
    axis.title.y = element_text(vjust = 3),  
    panel.grid = element_blank()  
  )  
print(p)
```



Excellent use of `ggplot()` to make a figure that looks like the clunky base graphics. Maybe you can improve on it in the homework assignment?

We can change the parameters of the standard normal to change both the location and the scale of our distribution. The influence of changing the mean, or average, on the location of a distribution is perhaps obvious. But, the influence of variance may be less intuitive, so let's have a look!

Create two more random samples, one with a larger `sd` and one with a smaller `sd`, to see how this changes the shape of the distribution:

```
samp2 <- rnorm(n = 1e4, mean = 0, sd = 2)
samp3 <- rnorm(n = 1e4, mean = 0, sd = .5)
```

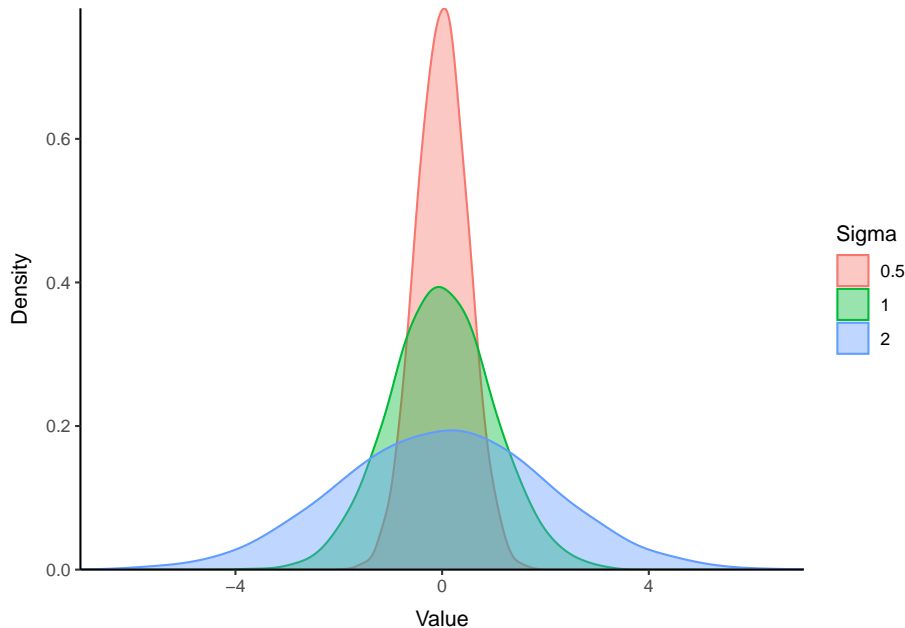
Let's put them in a data frame with `samp` so they're easy to plot. We combine all three random samples into one column called `Value`. Then, we create a column to hold the standard deviation used in each sample (`Sigma`). If we make that one into a factor, we can use the `Sigma` columns to plot the samples as separate lines by tweaking our plotting code.

```
normals <- data.frame(
  Value = c(samp, samp2, samp3),
  Sigma = factor(
    c(
      rep(1, length(samp)),
      rep(2, length(samp2)),
      rep(.5, length(samp3))
    )
  )
)
```

Next, we can just add these to the plot to compare the sampling distributions. This time, we tell R to fill the area under our lines based on sample ID with a default color scheme by saying `fill = Sigma` in our `ggplot()` call. We also added `color = Sigma` to make the lines the same default colors. Remember, you can specify your own.

```
p <- ggplot(data = normals,
  aes(x = Value, group = Sigma, fill = Sigma, color = Sigma)) +
  geom_density(adjust = 1.5, alpha = .4) +
  scale_x_continuous(limits = c(-7, 7), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab("Value") +
  ylab("Density") +
  theme_classic() +
  theme(
```

```
axis.title.x = element_text(vjust = -1),  
axis.title.y = element_text(vjust = 3),  
panel.grid = element_blank()  
)  
print(p)
```



The blue polygon in the plot above shows a distribution with greater variance than our z distribution (green). The red polygon shows a distribution with a smaller variance. Hopefully this helps demonstrate how variance influences the scale of the distribution.

5.4.1 The lognormal distribution

The **lognormal distribution** is a probability distribution that assumes our random variable is normally distributed on the **log scale**. This assumption allows us to incorporate **skew** into the normal distribution and change the location and scale of the normal distribution by transforming the parameters (μ and σ) onto the log scale. This is one of the more common data transformations that you will run into, e.g.: “We log-transformed the data to achieve normality...”. One of the other reasons for that is that all values (positive or negative) transformed from the log to the real scale are positive, so it helps prevent us from making negative predictions about phenomena or variables that can’t be less than zero.

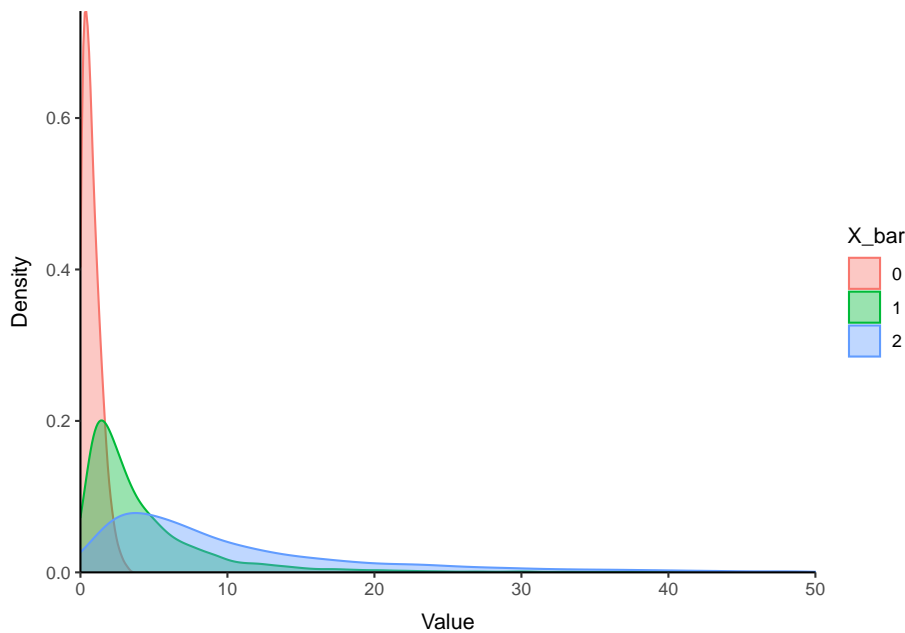
Let's take a look at how changes to the mean change the location of this distribution:

```
# Create random samples from log-normal
# distributions with different means
samp1 <- rlnorm(n=1e4, mean=0, sd=1)
samp2 <- rlnorm(n=1e4, mean=1, sd=1)
samp3 <- rlnorm(n=1e4, mean=2, sd=1)

# Put them in a data frame with the values
# of the means used to create them
lognormals <- data.frame(
  Value = c(samp, samp2, samp3),
  X_bar = factor(
    c(
      rep(0, length(samp)),
      rep(1, length(samp2)),
      rep(2, length(samp3))
    )
  )
)
```

Now you can plot these using the code above with a couple of modifications to show how the mean of the log-normal distribution influences the location.

```
p <- ggplot(data = lognormals,
  aes(x = Value, group = X_bar, fill = X_bar, color = X_bar)) +
  geom_density(adjust = 1.5, alpha = .4) +
  scale_x_continuous(limits = c(0, 50), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab("Value") +
  ylab("Density") +
  theme_classic() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```

You can see that the relative scale of these three distributions is similar, but the location shifts to the right on our x-axis as the value of \bar{X} (the mean) increases. Note also how this affects *kurtosis*.

5.4.2 The beta distribution

The **beta distribution** is a probability distribution that is constrained to the interval $[0, 1]$. But, it is incredibly flexible in its parameterization, and as a result is very useful for stochastic simulation of variables on the probability scale, such as survival.

The parameters of the beta distribution are α and β , or commonly **a** and **b** or **shape 1** and **shape 2** in R. Within this distribution, α pushes the distribution to the right (toward 1), and β pushes the distribution back toward the left (toward 0). The relative magnitude of α and β determine the location, shape, and scale of the probability distribution for our random variable. When α and β are equal, and greater than 1, the beta distribution looks like a normal distribution within the interval $[0, 1]$.

Let's take a look:

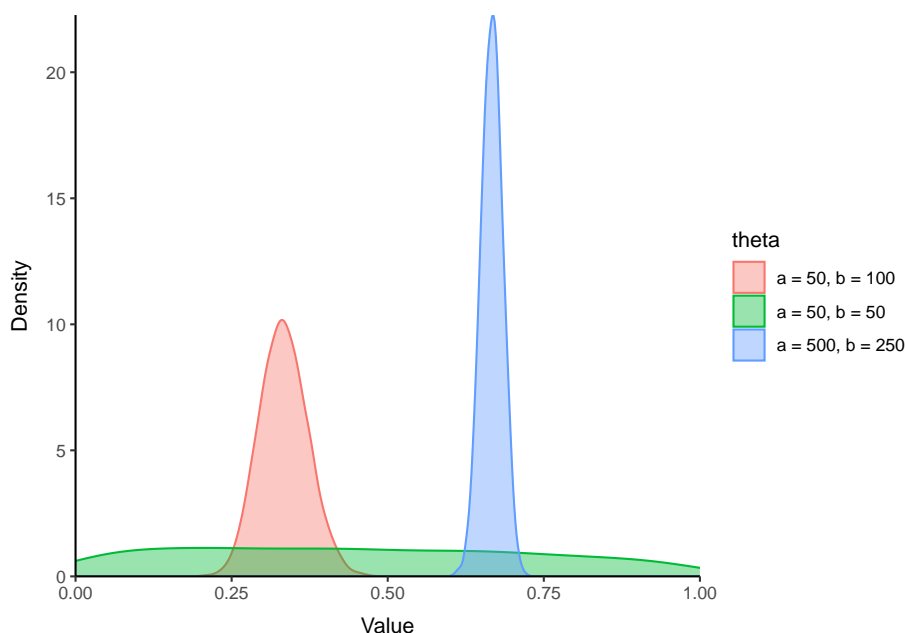
```
# Simulate random values from 3 different beta distributions
# so we can compare them
samp1 <- rbeta(n=1e4, shape1=50, shape2=50)
samp2 <- rbeta(n=1e4, shape1=50, shape2=100)
```

```
samp3 <- rbeta(n=1e4, shape1=500, shape2=250)

# Put them in a data frame with the values
# of the means used to create them. I am
# using "theta" because often that is how we
# refer collectively to a group of parameters
betas <- data.frame(
  Value = c(samp, samp2, samp3),
  theta = factor(
    c(
      rep('a = 50, b = 50', length(samp)),
      rep('a = 50, b = 100', length(samp2)),
      rep('a = 500, b = 250', length(samp3))
    )
  )
)
```

And then, we can plot them just like we did above. Copy and paste it - change what you need. Isn't code great?. Just don't forget to change the scale and the data in the plotting code!

```
p <- ggplot(data = betas,
            aes(x = Value, group = theta, fill = theta, color = theta)) +
  geom_density(adjust = 1.5, alpha = .4) +
  scale_x_continuous(limits = c(0, 1), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab("Value") +
  ylab("Density") +
  theme_classic() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```



Play around with these to see what kind of cool shapes you can make and where they are located within the range between zero and one.

5.5 Discrete distributions

Discrete probability distributions are useful for situations in which our random variable of interest can only take specific values within the interval of interest. For example, this might include age, counts, pass/fail, or any number of conceivable categories. As a result, these require a slightly different treatment of probability as a discrete, rather than continuous phenomenon. (Think back to our histogram that we started with in this chapter.)

5.5.1 Bernoulli

The **Bernoulli distribution** is a special case of the binomial distribution with a single trial (see below for clarification). Bernoulli outcomes are those for which the variable we are measuring can take on one of two values: a one or a zero. This distribution is useful for visualizing processes such as coin flips, yes/no responses, live/dead endpoints in lab studies, and a number of other very interesting phenomena. The Bernoulli distribution has a single parameter: the probability of success, but the number of successful outcomes is also governed by sample size: n , which R calls `size` because `n` was already taken.

We can simulate data from a Bernoulli distribution in one of two ways in R.

The “old-school” way of doing this was to draw from a binomial with a single **trial**. Here we randomly draw a single sample from a binomial with a single trial, and a 50% chance of success. We’ll use the example of hatching chicken eggs with some probability of success. If you are boring, you can think about flipping coins, too!

We’ll start with one chicken egg that has a 50% chance of successfully hatching (probability of success = 0.50).

```
rbinom(n=1, size=1, prob=.5)
```

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

There is also a function called `rbern` in the `Rlab` package that simplifies this for the specific case of a Bernoulli.

Let’s do it again with that function:

```
# Hatch one egg with 50% success rate
rbern(n = 1, prob = .5)
```

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

Or we could hatch a whole bunch of eggs:

```
# Hatch ten eggs, each with p = 0.5
rbern(n = 10, prob = .5)
```

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

Then, we could even count how many of those were successful. Do you remember how to do that? There are several different ways. You’ll have to come up with one for the homework assignment (hint: see Chapter 2).

5.5.2 Binomial

The **binomial distribution** is pretty similar to the Bernoulli distribution. In fact, the Bernoulli is just a special kind of binomial. The binomial includes a parameter called N (`size` in R) which corresponds to a number of trials per sample. We assume that this is 1 in the case of Bernoulli. In most cases in biology, it will suffice to use the Bernoulli, but for modeling we will want to understand the binomial for things like random stratified designs and nested

models that rely on the use of binomial distribution. Later in your career, you might even get into cool models that estimate N as a latent state to estimate population size (for example). Plus, using the binomial is way faster and can be more precise for certain regression applications [okay, that one should probably have a citation, but this is The Worst Stats Text ever, so go Google it].

To sample data from a binomial distribution, we can use `rbinom` from base R. In this example we tell R that we want 10 samples (`n`) from a binomial distribution that has 10 trials (`size`) and a probability of success (`prob`) of 0.5. This is like hatching ten eggs from each of ten chickens instead of just one chicken laying ten eggs.

```
# Take a random draw of 10 samples
# from a binomial distribution with 10 trials
# and probability of success equal to 0.50
rbinom(n=10, size=10, prob=0.5)
```

```
## [1] 4 5 5 7 6 3 5 5 5 3
```

Remember as you look through these that your numbers should look different than mine (at least most of the time) because these are being generated randomly.

5.5.3 Multinomial

The **multinomial distribution** is a further generalization of the Binomial and Bernoulli distributions. Here, there are one or more possible categorical outcomes (states), and the probability of each one occurring is specified individually **but all of them must sum to one**. The categories are, in this case, assumed to be a **mutually exclusive** and **exhaustive** set of possible outcomes.

We can use the multinomial distribution to randomly sample from categories (imagine our response variable is a categorical variable, like the names of the students in this class).

To do this, we need to read in the `s_names.csv` file from our `data` folder that is definitely in your working directory (**remember to set your working directory first**).

Read in the data file with `stringsAsFactors = FALSE` for purposes of demonstrating with categorical variables (not factors).

```
s_names <- read.csv('data/s_names.csv', stringsAsFactors = FALSE)
```

Next, let's assign the variable `name` in `s_names` to a vector for simplicity.

```
name <- s_names$name
```

Then, we can assign a uniform probability of drawing any given name if we divide one by the number of names.

```
# Calculate probability of getting a given
# name based on the length of the vector
prob_each <- 1 / length(name)

# Repeat this probability for each
# name in our vector of names
probs <- rep(prob_each, times = length(name))
probs
```

```
## [1] 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154
## [7] 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154
## [13] 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154
## [19] 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154
## [25] 0.03846154 0.03846154
```

This shows us that the probability of drawing any of the individual names is `{rprob_each}`.

Now, we can sample from a multinomial distribution using our objects. Here we are taking 5 samples from the distribution, each time we sample there is only one trial, and we are sampling with the 26 probabilities above.

Have a look:

```
rmultinom(n=5, size=1, prob=probs)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    1    0    0
## [2,]    0    0    0    0    0
## [3,]    0    0    0    0    0
## [4,]    0    0    0    0    0
## [5,]    0    0    0    0    1
## [6,]    0    0    0    0    0
## [7,]    0    0    0    1    0
## [8,]    0    0    0    0    0
## [9,]    0    0    0    0    0
## [10,]   0    0    0    0    0
## [11,]   0    0    0    0    0
## [12,]   0    0    0    0    0
```

```
## [13,] 1 0 0 0 0
## [14,] 0 0 0 0 0
## [15,] 0 0 0 0 0
## [16,] 0 0 0 0 0
## [17,] 0 0 0 0 0
## [18,] 0 0 0 0 0
## [19,] 0 0 0 0 0
## [20,] 0 0 0 0 0
## [21,] 0 0 0 0 0
## [22,] 0 1 0 0 0
## [23,] 0 0 0 0 0
## [24,] 0 0 0 0 0
## [25,] 0 0 0 0 0
## [26,] 0 0 0 0 0
```

WHOA a matrix??!!! **What does it all mean?**

Take a step back, breathe, and think about this. The rows in this matrix are you and your classmates. If we took one random sample from the multinomial distribution, it would look like this:

```
# Take a single sample from
# the list of student names
rmultinom(n=1, size=1, prob=probs)
```

```
##      [,1]
## [1,] 0
## [2,] 0
## [3,] 0
## [4,] 0
## [5,] 0
## [6,] 0
## [7,] 0
## [8,] 0
## [9,] 0
## [10,] 0
## [11,] 0
## [12,] 0
## [13,] 0
## [14,] 0
## [15,] 0
## [16,] 0
## [17,] 0
## [18,] 0
## [19,] 0
## [20,] 0
```

```
## [21,] 0
## [22,] 0
## [23,] 1
## [24,] 0
## [25,] 0
## [26,] 0
```

Here, we pulled a single sample from the distribution, and probability of sampling a given individual was 0.04 (1/26). If it makes it easier, we can put your names next to it:

```
cbind(name, rmultinom(n=1, size=1, prob=probs))
```

```
##      name
## [1,] "Ava"      "0"
## [2,] "Dillon"   "0"
## [3,] "Delaney"  "0"
## [4,] "Manolo"   "0"
## [5,] "Sarah"    "0"
## [6,] "Shannon"  "0"
## [7,] "Olivia"   "0"
## [8,] "Ebony"    "0"
## [9,] "Julia"    "0"
## [10,] "Davi"    "0"
## [11,] "Gabrielle" "0"
## [12,] "Jordan"   "0"
## [13,] "Tayler"   "0"
## [14,] "Summer"   "0"
## [15,] "Leah"     "0"
## [16,] "Christine" "0"
## [17,] "Ashley"   "0"
## [18,] "Katherine" "0"
## [19,] "James"    "0"
## [20,] "Emily"    "0"
## [21,] "Cassidy"  "0"
## [22,] "Maximillion" "0"
## [23,] "Sierra"   "1"
## [24,] "Kyle"     "0"
## [25,] "Diana"    "0"
## [26,] "Amanda"   "0"
```

Now, if I was calling on you randomly in class, after 10 questions, the spread of people who would have participated in class might look like this (or whatever you got - remember, it is random):


```
cbind(name, rmultinom(n=10, size=1, prob=probs))
```

```
##      name
## [1,] "Ava"      "0" "0" "0" "0" "1" "0" "0" "0" "0" "0"
## [2,] "Dillon"   "0" "1" "0" "0" "0" "0" "0" "0" "0" "0"
## [3,] "Delaney"  "1" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [4,] "Manolo"   "0" "0" "0" "0" "0" "0" "0" "0" "0" "1"
## [5,] "Sarah"    "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [6,] "Shannon"  "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [7,] "Olivia"   "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [8,] "Ebony"    "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [9,] "Julia"    "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [10,] "Davi"    "0" "0" "1" "0" "0" "0" "0" "0" "0" "0"
## [11,] "Gabrielle" "0" "0" "0" "0" "0" "1" "0" "0" "0" "0"
## [12,] "Jordan"  "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [13,] "Tayler"  "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [14,] "Summer"  "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [15,] "Leah"    "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [16,] "Christine" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [17,] "Ashley"  "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [18,] "Katherine" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [19,] "James"   "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [20,] "Emily"   "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [21,] "Cassidy" "0" "0" "0" "0" "0" "0" "0" "0" "1" "0"
## [22,] "Maximillion" "0" "0" "0" "1" "0" "0" "0" "0" "0" "0"
## [23,] "Sierra"  "0" "0" "0" "0" "0" "0" "1" "1" "0" "0"
## [24,] "Kyle"    "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [25,] "Diana"   "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [26,] "Amanda"  "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
```

Taking this one step further, we could just draw a name and stop looking at these ugly (no but really they are **awesome!**) matrices:

```
name[which(rmultinom(n=1, size=1, prob=probs)>0)]
```

```
## [1] "Amanda"
```

And now we have a way to randomly select an individual based on a multinomial distribution. What fun!

5.5.4 Poisson

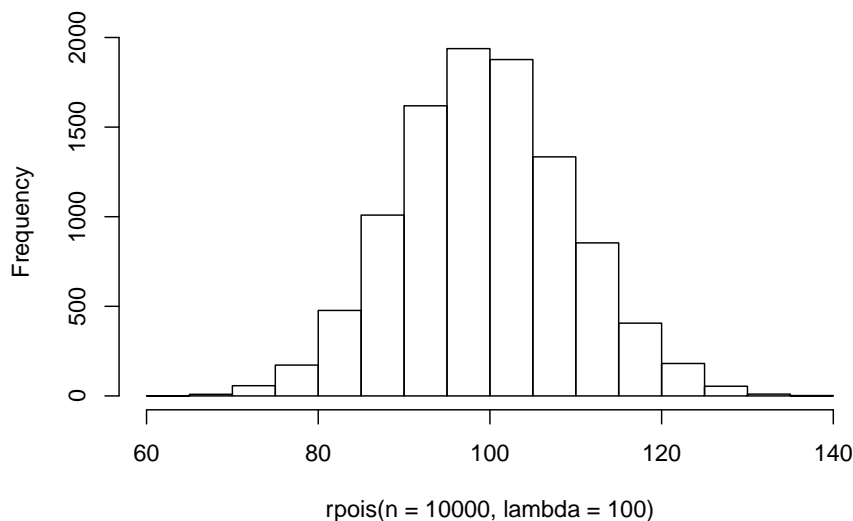
The **Poisson distribution** is used for counts or other integer data. This distribution is widely used (and just as widely misused!) for its ability to account

for a large number of biological and ecological processes in the models that we will discuss this semester. The Poisson distribution has a single parameter, λ , which is both the mean and the variance of the distribution. So, despite its utility, the distribution is relatively inflexible with respect to shape and spread.

Fun fact: this distribution was made widely known by a Russian economist to predict the number of soldiers who were accidentally killed from being kicked by horses in the Prussian army each year. It is named, however, after French mathematician Siméon Denis Poisson. [fails to provide citations for any of this]

Take a look at how the distribution changes when you change λ , and you will get an idea of how this one works. It is probably the most straightforward of any we've considered.

```
hist(rpois(n=1e4, lambda=100), main='')
```



We'll set it aside for now because it often fails us (or our data fail it, I suppose).

5.5.5 The negative binomial distribution

Okay, this one can be a little difficult to wrap your head around but it's an important one for us to know about. So, we will spend a little extra time setting this one up to try and be clear. Often, folks start out thinking that they're going to use a Poisson distribution and they end up collecting with data that do not conform to the relative inflexibility of that single-parameter

distribution. Where they end up usually tends to be a negative binomial in a best case (we'll talk about challenges associated with lots of zeros later in the book).

For the purpose of this class, we are not going to dive into the mechanics of the **negative binomial distribution**, but we do need to know what it looks like and why we might need it.

One useful way to conceptualize the negative binomial is “how long does it take for some event to occur?” For example, we might ask how long it takes a fish to start migrating, how long it takes a sea turtle to recover in a rehabilitation center, how long it will take for a terminal patient to expire (ooh, that's dark), or how frequently we see the expression of a gene of interest. These kinds of questions are asked in aptly named “time-to-event” models that rely on the variance structure of the negative binomial. In the context of these kinds of questions, the negative binomial is a discrete probability distribution (and not a continuous distribution) because the “time” component of the distribution is actually a series of independent Bernoulli trials (holy crap!). For example: if we want to know how many days it will take for a turtle to recover from an injury, what we are really doing is asking on each day until recovery, “Is today the day?”. Then, we flip a coin and find out. So, each day in this example is a Bernoulli trial. Another way to think about this is the number of failures occurring in a sequence before a target number of successes is achieved.

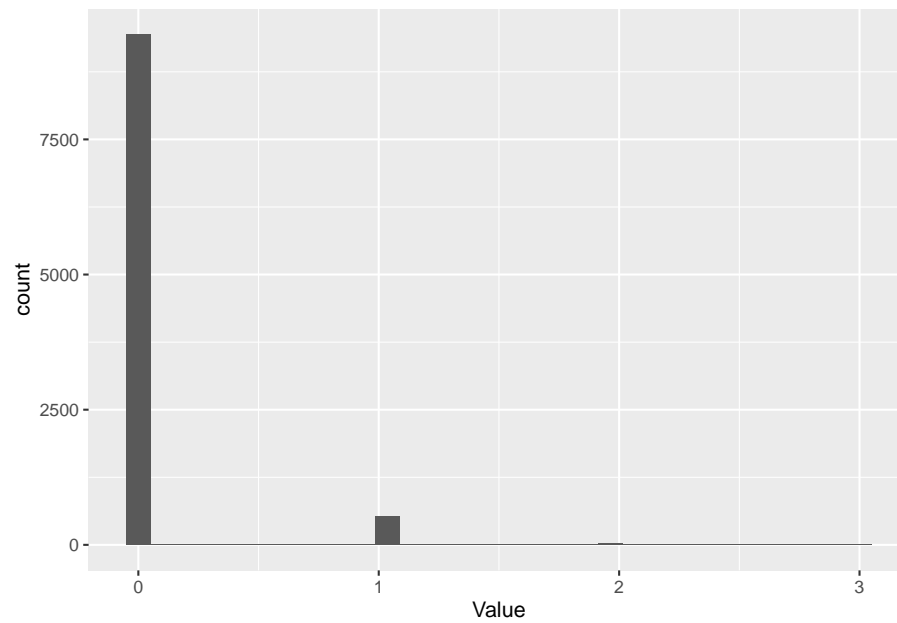
For the classical parameterization:

We will start by looking at how many failures are observed before one success in a sequence of Bernoulli trials.

With probability of success equal to 0.95, it doesn't take long and most of the probability mass is near zero, with a couple of stragglers further out.

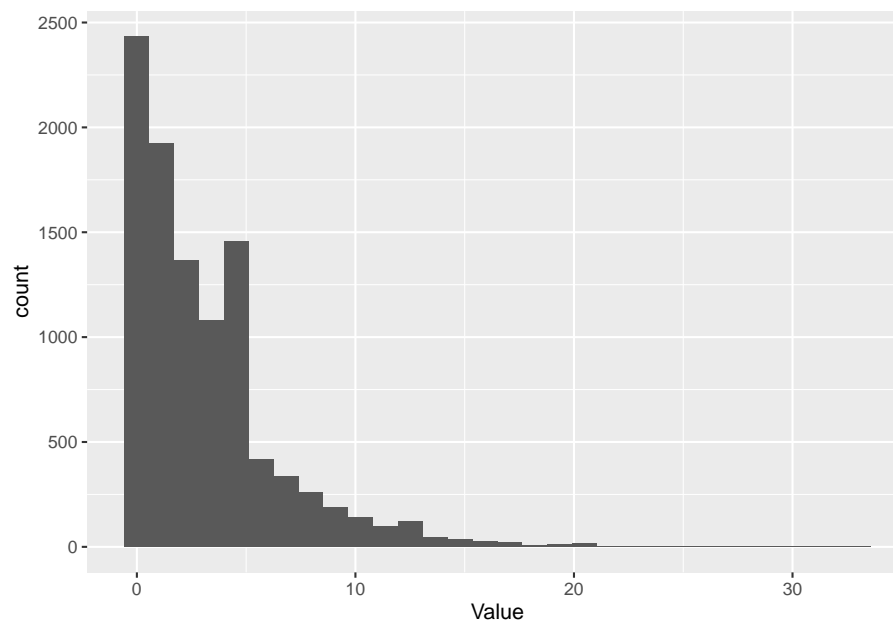
```
# Take a random sample from the negative binomial
Value <- rnbinom(1e4, size=1, prob=.95)

# Make a histogram of it with ggplot
ggplot() + geom_histogram( aes(x = Value) )
```



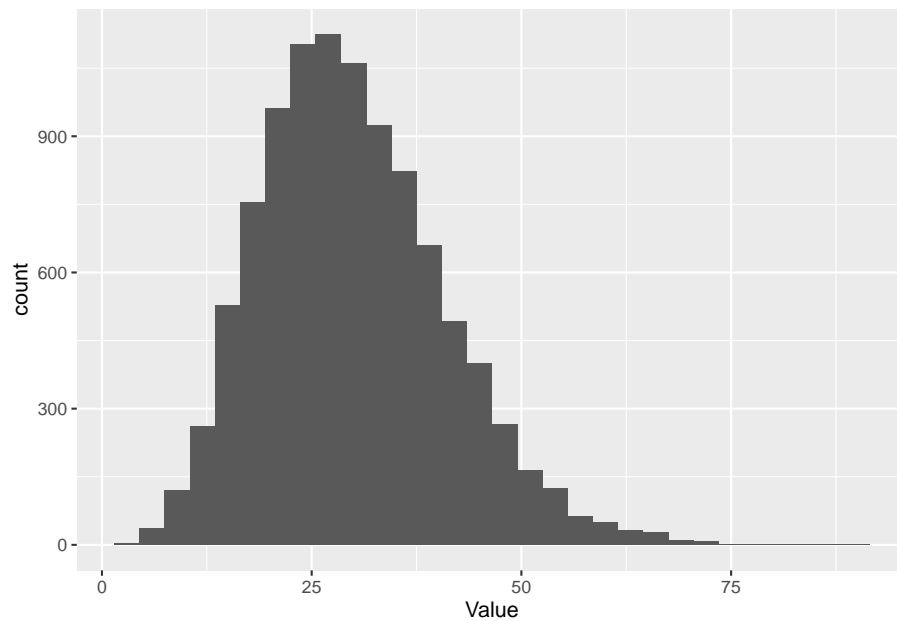
If we decrease probability of success in each trial to 0.25, we see more failures on average before we reach success. Most of the time, it still takes less than 5 trials to reach a success, but some times it takes much longer.

```
# Take a random sample from the negative binomial  
Value <- rnbino(1e4, size=1, prob=.25)  
  
# Make a histogram of it with ggplot  
ggplot() + geom_histogram( aes(x = Value) )
```



And, if we increase the number of successes that we use for our criterion, or target, then it spreads the distribution out even further.

```
# Take a random sample from the negative binomial  
Value <- rnbinom(1e4, size=10, prob=.25)  
  
# Make a histogram of it with ggplot  
ggplot() + geom_histogram( aes(x = Value) )
```

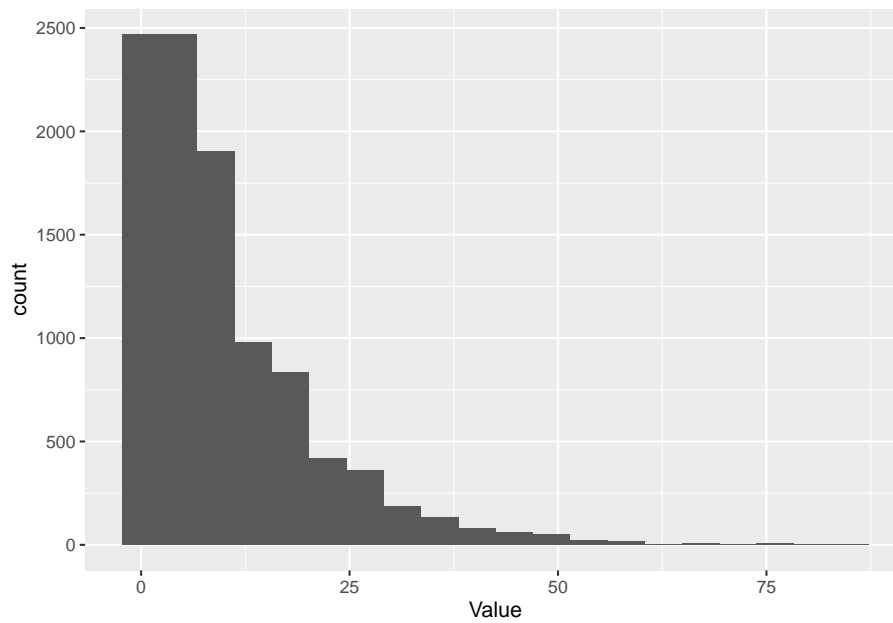


Now, because of its properties, the negative binomial is also useful for number of other applications that have nothing to do with interpreting the results of repeated binomial trials. Specifically, it has been widely used to represent Poisson-like processes in which the mean and variance are not equal (e.g., **overdispersion**). This has seen a lot of application in the field of ecology, especially for overdispersed count data.

Here, we draw 10,000 random samples from a negative binomial distribution with a mean of 10 and an overdispersion parameter of 1. The overdispersion parameter is called ‘size’ because this is an alternative parameterization that is just making use of the relationships between existing parameters of the negative binomial. It’s easy to grasp how the mean changes the location of the distribution.

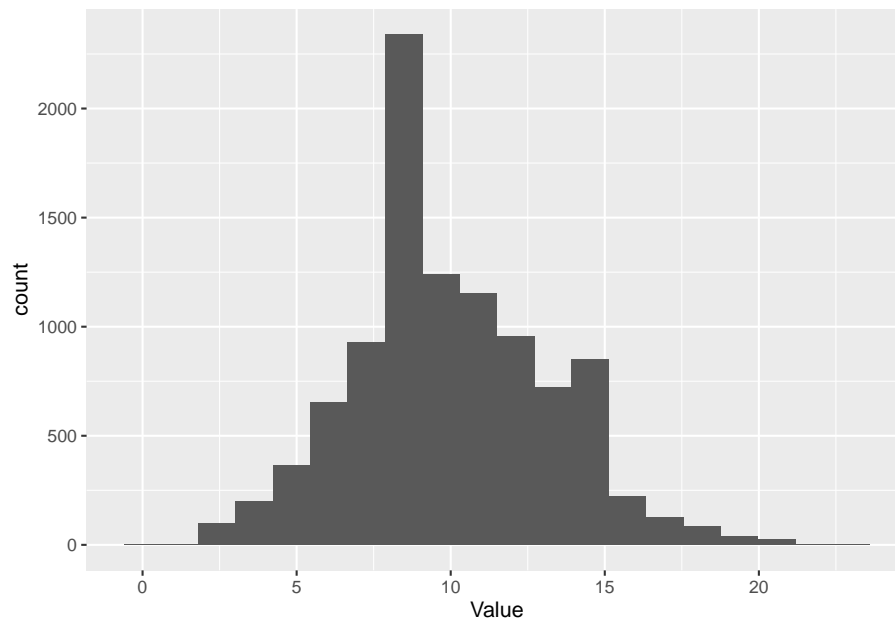
```
# Take a random sample from the negative binomial
Value <- rnbinom(1e4, mu = 10, size = 1)

# Make a histogram of it with ggplot
ggplot() + geom_histogram( aes(x = Value), bins = 20 )
```



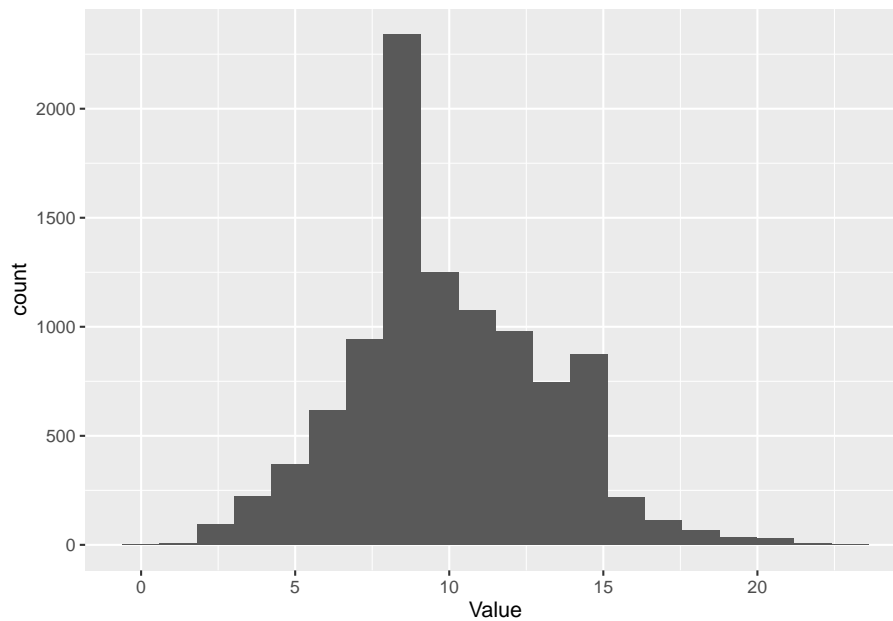
But, note how the overdispersion parameter changes things if you run the following code:

```
# Take a random sample from the negative binomial  
Value <- rnbinom(1e4, mu = 10, size = 1000)  
  
# Make a histogram of it with ggplot  
ggplot() + geom_histogram( aes(x = Value), bins = 20 )
```



A more intuitive way (I think) to work with the negative binomial in R is by using the `MASS` package. In this parameterization, we use the mean and the dispersion parameter explicitly so it makes more sense:

```
# Take a random sample from the negative binomial  
Value <- rnegbin(1e4, mu = 10, theta = 1000)  
  
# Make a histogram of it with ggplot  
ggplot() + geom_histogram( aes(x = Value), bins = 20 )
```

The results are pretty much identical. Just two different naming systems for the parameters.

5.6 Sample statistics

In this section, we will learn how to derive the parameters of the **normal distribution** using a few different methods in R. We will use this opportunity to re-introduce the parameters as **moments** of the distribution so we can talk about what we mean by **confidence intervals**. We also will introduce a couple of different methods for calculating moments of a distribution. Specifically, we will look at how to derive...

5.6.1 Moments about the mean

Sounds fancy, huh? Here they are, like a bandaid:

1. Zeroth moment

- This is the sum of the total probability of the distribution 1.00, always

2. First moment

- The mean
- We will look at a few ways to calculate this

3. Second moment

- The variance
- As with the mean, we will examine a couple of options for calculating

4. Third moment

- Skew
- We won't calculate for this class, but we have discussed, and this parameter contributes to the location/spread of the distribution (how far left or right the peak is)

5. Fourth moment

- Kurtosis
- Similarly, we won't cover the calculation, but this is another moment that we may have discussed with respect to departure from a z distribution in the normal

5.6.2 Estimating parameters of the normal distribution from a sample

The tools demonstrated below can be used for most of the probability distributions that have been implemented in R, and we could go on and on forever about them. But, for the sake of our collective sanity we will walk through the tools available using the normal distribution alone. Most of the time this will suffice because our objective in understanding other distributions is really just so that we can use them to assume asymptotic normality in response variables (with transformations) or parameter distributions (with link functions) later on anyway.

5.6.2.1 Method of moments estimator

The moments of the normal distribution are well defined, and you are probably familiar with how to calculate a mean (average) already. See if you can rearrange this in a way that makes sense with how you know to calculate a **mean** and a **variance**!

Start by simulating a variable with a known mean and standard deviation. We'll pretend that we are simulating cold temperatures here:

```
# Take a random sample from a normal
# with a mean of 20 and a standard
# deviation of 2
test_norm <- rnorm(1e4, 20, 2)
```

First, we'll estimate it by making our own function:

```
# Write the function
# First, define a function by name
norm.mom = function(x){

  # Calculate mean
  x_bar = (1/length(x)) * sum(x)

  # Calculate variance
  sigma2 = (1/length(x)) * sum((x-x_bar)^2)

  # Return the calculations
  return(c(x_bar, sigma2))
}

# Test the function
norm.mom(test_norm)
```

```
## [1] 19.970094  3.990084
```

Because this one is so common, R has built-in estimators that rely on the exact solution provided by the formulas for the first two moments of the normal distribution:

```
mean(test_norm)
```

```
## [1] 19.97009
```

```
var(test_norm)
```

```
## [1] 3.990483
```

Wow, that was a lot less code. That is the beauty of having these functions available. How do these compare to the answers returned by our function if you scroll back up?

5.6.2.2 Maximum likelihood estimator

R also has built-in **maximum likelihood** estimators that provide an exact solution to the first two moments of the normal distribution. These are available through the MASS package.

```
fitdistr(test_norm, 'normal')
```

```
##          mean          sd
## 19.97009390  1.99751941
## ( 0.01997519) ( 0.01412460)
```

Only one problem here: R doesn't report the second moment! It reports the square root of the second moment: the **standard deviation**!

Finally, let's write our own function and maximize the likelihood with the `optim()` function in R.

```
# Define the function
normal.lik = function(theta, y){

  # The starting value for
  # mu that we provide
  mu = theta[1]

  # The starting value for
  # sigma2 that we provide
  sigma2 = theta[2]

  # Number of observations in the data
  n = nrow(y)

  # Compute the log likelihood of the
  # data (y) using the likelihood
  # function for the normal distribution
  # given the starting values for our
  # parameters (contained in the vector 'theta')
  logl = -.5*n*log(2*pi) -.5*n*log(sigma2)-(1/(2*sigma2))*sum((y-mu)**2)
  return(-logl)
}
```

Now, we use the `optim` function to maximize the likelihood of the data (technically by minimizing the $-2 \times \log[\text{likelihood}]$) given different values of our parameters (`mu` and `sigma2`).

To get started, we need to take a guess at what those parameters could be. (Yes, we know they are $\mu = 20$ and $\text{sd} = 2$)

```
optim(c(20, 4), normal.lik, y=data.frame(test_norm))
```

```
## $par
```

```
## [1] 19.969855  3.991037
##
## $value
## [1] 21108.45
##
## $counts
## function gradient
##      47      NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

The pieces are in `pars` here (right where we told R to put them!). We can also make the output into an object and call the parts by name:

```
# Make it into an object
est = optim(c(0, 1),
            normal.lik,
            y=data.frame(test_norm)
            )
```

```
## Warning in log(sigma2): NaNs produced
## Warning in log(sigma2): NaNs produced
## Warning in log(sigma2): NaNs produced
## Warning in log(sigma2): NaNs produced
## Warning in log(sigma2): NaNs produced
## Warning in log(sigma2): NaNs produced
```

Look at the structure I'll be damned, it's a list! Hey, we learned about those!

```
str(est)
```

```
## List of 5
## $ par      : num [1:2] 20 4
## $ value    : num 21108
## $ counts   : Named int [1:2] 97 NA
```

```
##    ..- attr(*, "names")= chr [1:2] "function" "gradient"
##    $ convergence: int 0
##    $ message      : NULL
```

And, here are the estimates:

```
# Both
est$par
```

```
## [1] 19.967219  3.995283
```

```
# The mean
est$par[1]
```

```
## [1] 19.96722
```

```
# The variance
est$par[2]
```

```
## [1] 3.995283
```

There you have it, a couple of different ways to calculate parameters of the normal distribution using a couple of different approaches each.

5.6.3 Quantiles and other descriptive statistics

There are a number of other ways we might like to describe this this (or any) sampling distribution. Here are a few examples that we will work with this semester.

```
# Here is the median, or 50th percentile
median(test_norm)
```

```
## [1] 19.96312
```

```
# The 95% confidence limits
quantile(test_norm, probs = c(0.025, 0.975))
```

```
##      2.5%      97.5%
## 16.03437 23.89289
```

```
# Interquartile range (Q1 and Q3)
quantile(test_norm, probs = c(0.25, 0.75))
```

```
##      25%      75%
## 18.60878 21.31663
```

```
# Range of sample
range(test_norm)
```

```
## [1] 12.40963 27.28385
```

5.7 Next steps

Here, we have explored some of the probability distributions that we use to describe samples (data) that we collect from the real world. In Chapter 6 we will explore how these sampling distributions can be used for statistical inference before diving into applied statistical analyses for the remainder of the book. Hopefully the order of things is starting to make some sense! If not, well...this *is* The Worst Stats Text ever.

Chapter 6

Inferential statistics

These are hot peppers. Like hot peppers, statistics can cause pain and heartburn if you are not accustomed to them. Ready or not, let's get munching!!

This week we will begin conducting our first statistical tests! We are going to start small and simple, and we will build complexity during the remainder of the semester. We will also start to make more use of some of the programming techniques that you have been developing, and we will build a foundation for moving into regression models in coming weeks.

We'll start with some simple methods for testing hypotheses about sampling distributions this week. Although relatively limited in scope within the fields of biology and ecology, these tend to be fairly robust tests, and can be powerful tools if studies are designed thoughtfully. For this week, we will focus on implementation of one-sample t-tests, two-sample t-tests, Wilcoxon tests, and frequency analysis using a χ^2 test. Within the context of the assumptions of these tests we will also discuss the F-test and the Shapiro-Wilk test of normality. In short, you probably will learn more statistical tests in this preliminary chapter about statistical inference than you have in your college career to this point. Take your time and soak in all the mathy goodness. We'll need it!

For this Chapter, we will continue working with packages from the **tidyverse**. You can go ahead and put this in the top of your code for the chapter if you want to load it all at once:

```
library(tidyverse)
```

We will also need the grass carp data for this exercise, which we will load from **grasscarp.csv**. Remember that you can download all of the class data [here](#) or you can get the individual **grasscarp.csv** file by clicking [here](#) and saving with **Ctrl + S** (Windows) or **Command + S** (Mac OS-X).

These data come from a long-term study of fish population responses to changes in their primary food source, the invasive hydrilla (*Hydrilla verticillata*). There are a whole bunch of columns in here! The important variables for this chapter are **Year** (year of fish collection), **Age** (the age of each fish), **Length** (total length of fish in mm), and **hydrilla** (hectares of hydrilla measured each Year).

6.1 One-sample tests

Sometimes, we are interested in simply knowing whether or not the measurements we've obtained from an individual or a group are representative of a larger population. For example, we may have a 'control' group in an experiment and we want to know if the group is truly representative of the population average or some measurement we have collected from a different biological population. For these situations, we will rely on one-sample tests this week and we'll look at other (totally related) options moving forward.

6.1.1 One sample t-test

We will examine parametric and non-parametric examples of one-sample tests here to demonstrate why and how we use them.

Let's start with a simple example of how we might do this, and what the results actually mean. We'll use some data from grass carp (*Ctenopharyngodon idella*) from Lake Gaston, Virginia and North Carolina, USA for this example. We will compare the size of grass carp at specific ages with their population density using a few different tools

Read in the data set:

```
grasscarp <- read.csv('data/grasscarp.csv')
```

Just for funsies, you could also read this in directly from the link to the raw data in the GitHub repository for this book if you have an internet connection:

```
grasscarp <- read.csv('https://raw.githubusercontent.com/danStich/worst-r/master/data/g')
```

Remember to check out the data set in your Environment tab so you understand how many observations there are and how many variables (as well as their types).

Let's start by asking a simple biological question: is the size of age-3 grass carp different from the average size of fish in this population?

First, let's create a sample that includes only age-3 fish. We will store this to a new vector called `age3_lengths`.

```
age3_lengths <- grasscarp$Length[grasscarp$Age == 3]
```

Now, let's compare the `Length` of age-3 fish to the rest of the population using a one-sample t-test. To do this, we need to pass `age3_lengths`

```
# Run the test and save the output to an object
our_test = t.test(age3_lengths,
                  mu = mean(grasscarp$Length),
                  conf.level = 0.95
                )

# Print the results of the object to the console
print(our_test)

##
## One Sample t-test
##
## data:  age3_lengths
## t = -18.829, df = 47, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 973.0118
## 95 percent confidence interval:
##  749.2547 792.4536
## sample estimates:
## mean of x
##  770.8542
```

Okay, so what does that mean???

First, let's look at what we've done here.

We've conducted a one-sample t-test.

The null hypothesis was that:

(H_0) : the sample (age-3 fish) did not differ in `Length` from the mean of the population.

This is because we stated no specific alternative hypothesis when we executed the t-test above. If we had used a different alternative hypothesis (i.e. `greater` or `less` in the argument `alternative`) then our null would be formalized as: "The length of age-3 fish is not significantly greater (or less) than the population mean".

Finally, we specified the confidence level. Here, we are told R that we want to know the result with a confidence level of 95% (0.95). This corresponds to a

Type-I error rate (α) of 0.05. This means we are looking for $p < 0.05$ to conclude that the sample is statistically different from the population mean. Since $p < 0.001$, we reject the H_0 and conclude that age-3 fish are significantly shorter than the population mean.

6.1.1.1 Output

R returns the output of statistical tests as objects, and you can reference any part of those objects by name or index. The type of object that R returns, and how you access the parts depends on the type of test you ran and with what options.

Like so many other models objects, our one sample t-test is stored as a list:

```
str(our_test)

## List of 10
## $ statistic : Named num -18.8
## ..- attr(*, "names")= chr "t"
## $ parameter : Named num 47
## ..- attr(*, "names")= chr "df"
## $ p.value    : num 1.57e-23
## $ conf.int   : num [1:2] 749 792
## ..- attr(*, "conf.level")= num 0.95
## $ estimate   : Named num 771
## ..- attr(*, "names")= chr "mean of x"
## $ null.value : Named num 973
## ..- attr(*, "names")= chr "mean"
## $ stderr     : num 10.7
## $ alternative: chr "two.sided"
## $ method     : chr "One Sample t-test"
## $ data.name  : chr "age3_lengths"
## - attr(*, "class")= chr "htest"
```

We can just look at the names if there are specific pieces in which we are interested. For example, we might want to save the p-value (`p.value`):

```
# Shows us the names of the things inside the model list
names(our_test)
```

```
## [1] "statistic" "parameter" "p.value"    "conf.int"   "estimate"
## [6] "null.value" "stderr"     "alternative" "method"     "data.name"
```

```
# This stores our p-value to an object for use
p_out = our_test$p.value

# And of course we can look at it
print(p_out)
```

```
## [1] 1.572767e-23
```

Now we can go through the output as it is displayed by:

```
# Print a summary of the test
print(our_test)
```

```
##
## One Sample t-test
##
## data: age3_lengths
## t = -18.829, df = 47, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 973.0118
## 95 percent confidence interval:
## 749.2547 792.4536
## sample estimates:
## mean of x
## 770.8542
```

The first line of the output gives us the actual data that with which we are working- nothing of interest here other than a quick sanity check until later on in the course.

The second line shows the ‘statistics’ that we are interested in: **t** is the calculated value of the test statistic for the t-test in this case. The **df**, or degrees of freedom, is the number of observations in the sample, minus the number of parameters that we are estimating (in this case, just one: the mean). Our **p-value** is the probability of observing data that are more extreme than what we observed if the null hypothesis is in fact true (i.e. the probability that rejection of the null is inappropriate). Again, because it is smaller than α we reject the null and accept the alternative hypothesis.

Our **alteranive hypothesis** (H_A) was that the sample mean is not equal to population mean. We can specify other alternatives (and therefore nulls) in this and other models in R.

Finally, R reports the mean and the 95% confidence interval of **age3_lengths**.

6.1.1.2 Assumptions

It's always important for us to think about the assumptions that we are making when (read *before*) conducting a statistical test. First, there are implicit assumptions that we make. For example, we assume that the data are representative and were collected in a random manner in this case. Then, there are explicit assumptions that we make in specific tests.

For the one-sample t-test, the assumption that we really care about is:

1. The data are normally distributed

The t-test is generally robust to violations of this assumption provided that sample sizes are large enough (Google Central Limit Theorem, this is The Worst Stats Text ever). But, it is always good to check. In particular, when we are working with small sample sizes like this example ($n = 48$), we should really make sure that things look okay.

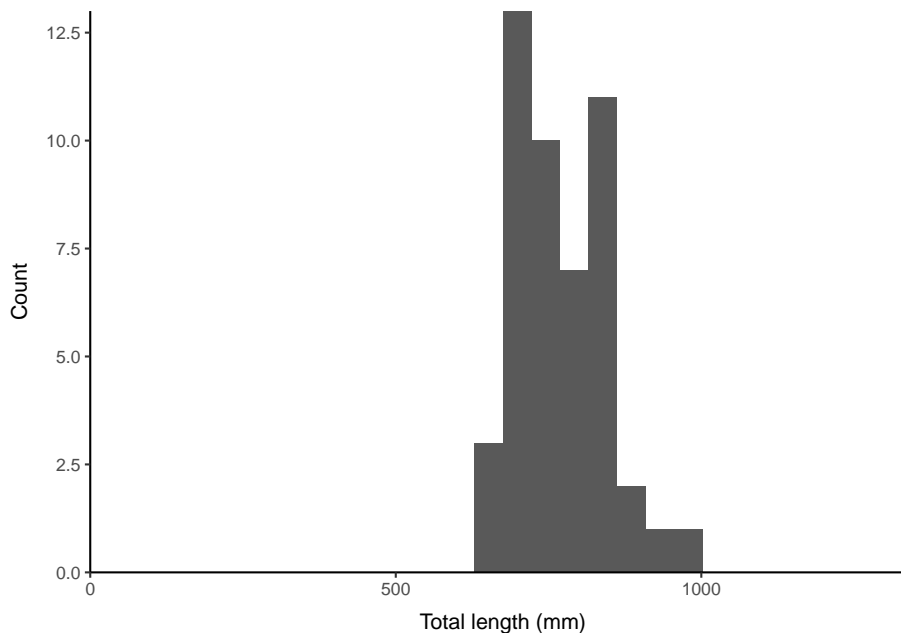
6.1.1.3 Checking assumptions

Visual check for normality

One simple way to gauge assumptions of normality is to look at a plot of the data. As you will see later, we are usually concerned with the *residuals*, but we can look at the actual data here because we have only one group and if it's normal so are its errors.

Have a quick look at these to see what we are working with using the histogram code from Chapter 4. I set the x-axis limits below using the maximum `Length` from the `grasscarp` data so we can see what part of the length range we've sampled here.

```
p <- ggplot() +
  geom_histogram(aes(age3_lengths), bins = 30) +
  scale_x_continuous(limits=c(0, max(grasscarp$Length)), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab("Total length (mm)") +
  ylab("Count") +
  theme_classic() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```



Sure, looks totally normal to me? Wouldn't it be great if there were a statistical test for determining whether this sample is different from the normal? Great that you should ask.

6.1.1.3.1 Tests of normality (Shapiro-Wilk) The Shapiro-Wilk test is commonly used to test normality of a distribution as a check of assumptions. We can use this to test whether our data deviate from normal in the following manner:

```
shapiro.test(age3_lengths)
```

```
##  
## Shapiro-Wilk normality test  
##  
## data:  age3_lengths  
## W = 0.95715, p-value = 0.07746
```

First, note that the test statistic is the W statistic for this test.

Second, we have a p-value of 0.0774637. **Oh, no!** Wait, what does that mean?

For this test, we actually don't want $p < 0.05$ if we are relying on assumptions of normality, so this is a "good" thing. But, it doesn't necessarily mean `age3_lengths` is normally distributed. It just means that we can't tell if the sample we have collected is different from normal (we fail to reject the null but

can't "accept" it). I guess that is good enough, but that p-value is awfully close to 0.05 for my taste.

So, are we up that proverbial tributary without a paddle, or can we salvage the mess and move on with life? Don't worry, there's a statistical test for that, too.

6.1.2 Wilcoxon test

You can think of the Wilcoxon test as a non-parametric analog of the t-test. In general, non-parametric tests tend to be slightly more "conservative" than the parametric alternatives because they require fewer assumptions. However, non-parametric tests can be useful where our data are not normal, or we don't feel we have sufficient data to know (hmm...maybe don't conduct any tests, in that case!).

Here, we are checking for shifts in the median (not the mean) of one or more samples.

Why is this? The mean of a non-normal distribution is not always a useful descriptor of the probability mass under a distribution (it still describes 'central tendency' but does not necessarily describe the place where 'most of the data are'). But, the median always (as in always, always, always) describes central tendency of the data, so we can pretty much use it for describing any sample. This is because the median is defined as the "middle" value. That is, half of the data should fall on either side of the median if you lined up all of your data on the equator (wrong metaphor?).

...back to the Wilcoxon test.

```
# First, do this and have a quick read:
?wilcox.test
```

Now we can actually run a test to see if the median length of age-3 fish is statistically different from the median value of length in the sample.

```
wilcox.test(
  age3_lengths,
  mu = median(grasscarp$Length),
  alternative = 'less',      # Note different alternative here!
  exact = FALSE             # Not much data, so not exact
)

##
## Wilcoxon signed rank test with continuity correction
##
## data:  age3_lengths
```



```
## V = 0, p-value = 8.414e-10
## alternative hypothesis: true location is less than 1007
```

Interpreting the results is essentially the same as for the t-test, but without the degrees of freedom, so we won't belabor this. Importantly, the test, being robust to any distributional assumptions, should also (and does) tell us that the length of age-3 fish is significantly shorter than the population mean (or median- whichever you used).

6.2 Two-sample tests

Okay, with that out of the way, now we can get on to some tests that might be a little more meaningful to most people. We can use **two-sample** tests to whether two groups differ in some metric of interest. This lends itself naturally to use in controlled experiments that we conduct in laboratories, for example.

6.2.1 The two-sample t-test

If you have been exposed to only one statistical test already it is probably the two-sample t-test. This is a test that is used to test for differences in some continuous variable between two groups. The test statistic itself is pretty trivial to calculate. You can find a video of that here. **Seriously, if you have never done a t-test, watch the 6-minute video now.** Otherwise, you may not understand what follows. I am not going to go into the math here because this is The Worst Stats Text ever. The video will also help you understand how ANOVA and other tests work later. Understanding how these tests work will give you phenomenal cosmic powers when it comes to analyzing biological data. If you email asking me how a t-test works, I am going to send you this video.

Let's keep working with the **grasscarp** data for now for the sake of consistency. But, now we want to know if there is a difference in mean length of fish depending on whether their population density is high or low. To look at this, we'll need to make some groups in our **grasscarp** data that correspond to years of high and low density.

You can compare fish density between years quickly using the summary pipeline demonstrated in Chapter 3

```
grasscarp %>%
  group_by(Year) %>%
  summarize(dens = unique(nha))
```

```
## # A tibble: 5 x 2
```

```
##      Year  dens
##    <int> <int>
## 1  2006    21
## 2  2007    58
## 3  2009    44
## 4  2010    43
## 5  2017   127
```

You can see that density was much higher in 2017 than in any of the preceding years. This is because hydrilla area was reduced by several hundred hectares (ha) between 2010 and 2014 (which was actually the reason we went out to collect more data in 2017). But, these are just means and we need to be able to account for the variability in these measurements to call it science.

So, let's build some groups based on high and low density years. First, we'll add a new categorical variable to `grasscarp` called "density", and we'll fill it all in with the word "low" because there is only one year when density was high.

```
grasscarp$density <- "low"
```

Next, we'll change all of the observations for 2017 to "high" so we have low density and high density groupings in our `density` column. This way, we only have to change the variable for one year.

```
grasscarp$density[grasscarp$Year == 2017] <- "high"
```

Then, we'll subset the data to look at a single age so our comparisons are fair between years. I picked `Age == 10` because 10 years is in the middle of the range of ages in the data set. You can try it with another age as long as there are enough data.

```
mid_carps <- grasscarp %>% subset(Age == 10)
```

Now, we can conduct our two-sample t-test!

The syntax is pretty straightforward, and is similar to what we used above, except that now we have two groups so we will omit `mu` and specify the t-test as a formula with independent (x, `density`) and dependent (y, `Length`) variables. There is no pairing of our observations, so we specify `paired = FALSE`, and we tell R we don't want to assume that the variance of `Length` is equal between `density` groups.

```
t.test(Length ~ density,
       data = mid_carps,
       paired = FALSE,      # 2-sample test, not "paired")
```

```

    var.equal = FALSE,    # We make no variance assumption
    conf.level = 0.95     # Alpha = 0.05
  )

##
##  Welch Two Sample t-test
##
## data:  Length by density
## t = -3.2263, df = 11.133, p-value = 0.007952
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -190.03710  -36.03433
## sample estimates:
## mean in group high  mean in group low
##           987.7143           1100.7500

```

The interpretation of the results is much the same as with the one-sample t-test, except that we are now testing the null hypothesis that there is no difference between groups.

We reject the null hypothesis, and we conclude that age-10 fish were significantly larger during periods of low population density than they were during years of high population density ($t = -3.2262903$, $df = 11.1326183$, $p < 0.05$). Makes perfect sense!

6.2.1.1 Assumptions

Equal variance

Now that we are using two samples, we should be cognizant that this test assumes equal variances in the independent variable between our two groups. If our variances are not equal, then we need to account for that (R actually assumes that the variances are unequal by default).

Let's test to see if the variances were equal between age-10 fish in the **high** and **low** density groups. To do this, we will conduct an F-test on the ratio of the two variances. If the ratio of the variances is different than one, we reject the null that the variances are the same.

```
var.test(Length~density, mid_carps)
```

```

##
##  F test to compare two variances
##
## data:  Length by density

```

```
## F = 0.72988, num df = 20, denom df = 7, p-value = 0.5423
## alternative hypothesis: true ratio of variances is not equal to 1
## 95 percent confidence interval:
##  0.1634026 2.1950440
## sample estimates:
## ratio of variances
##           0.729877
```

Wow, this is way to easy. I hope that you are beginning to understand the **GLORY OF R**. This test could be a real pain in other software programs, and may not even be an option in many.

Back on topic...we fail to reject the null hypothesis that the variances were equal. In this case, we now feel validated in the use of a two-sample t-test regardless of what R uses as the default (yes, sarcasm intended).

6.2.1.1.1 Normality Yes, we are still worried about this one because of the reasons given in the previous section. We can check this the same way as before. End of section.

6.2.2 Two-sample Wilcoxon test

If we were in violation of normality, we would use the Wilcoxon test to test for differences in ranks. I will not go through the whole thing again here. As with the t-test, if you have not been exposed to doing a rank-sum test by hand you really should **watch a video of how to do it**. It really is easy once you've seen it and the video demystify the test for you.

I will note that the syntax is very much the same to that of the t-test now. This will pretty much stay the same for the next 6 chapters. Thank R, not me.

```
wilcox.test(Length~density, mid_carps)
```

```
##
## Wilcoxon rank sum test with continuity correction
##
## data: Length by density
## W = 26, p-value = 0.005015
## alternative hypothesis: true location shift is not equal to 0
```

As expected, this test also shows that the two samples differ significantly.

Note: this is equivalent to the Mann-Whitney U-test you may have learned about elsewhere. Had these samples been paired, R would have defaulted to a signed-rank test, with which you may also be familiar.

6.2.3 Presenting your results

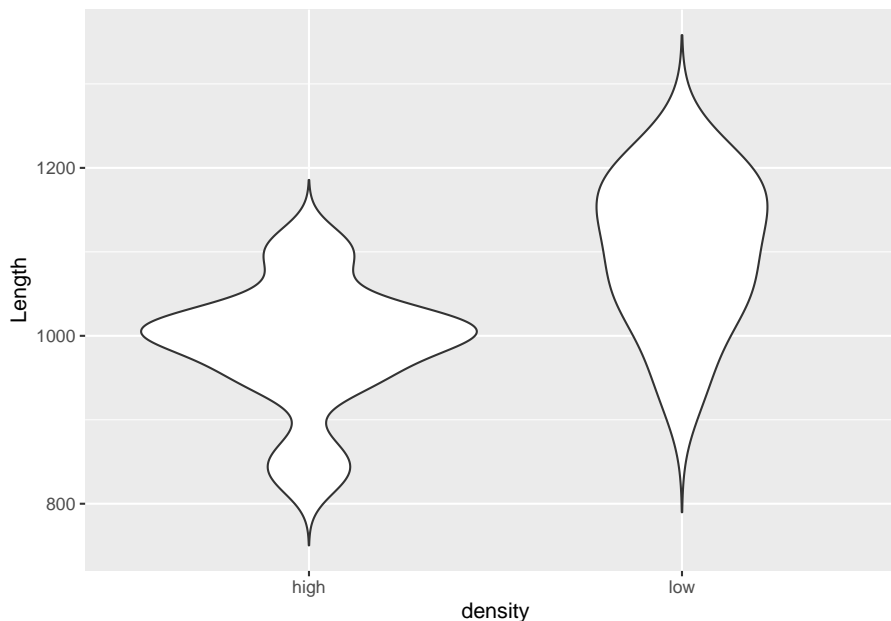
While it is important to report the test statistics, `df`, etc., it can be just as meaningful to give the sample means (reported in the `t.test`) and show a graph. **Remember:** don't make shitty graphs. Be proud of your results and show your readers what they mean.

In this case, a boxplot or a violin plot would work great. We haven't looked at violin plots yet, so let's give them a whirl!

Violins are a lot like box plots except they give us a little better visual description of the shape of sampling distributions within groups. I added some `ggplot2` functions to control fill and color of the violins in the example below. You can check out this [blog post](#) for some other cool examples with other `ggplot` geometries. Play around with the plotting code above to change what you like. Remember, all of the customization achieved using the `theme()` function is the same across plot types.

Here is a quick, ugly violin plot with some basic options. Pretty easy to make, but also kind of makes you want to puke.

```
ggplot(mid_carps, aes(x = density, y = Length)) +  
  geom_violin(aes(group = density), trim = FALSE)
```

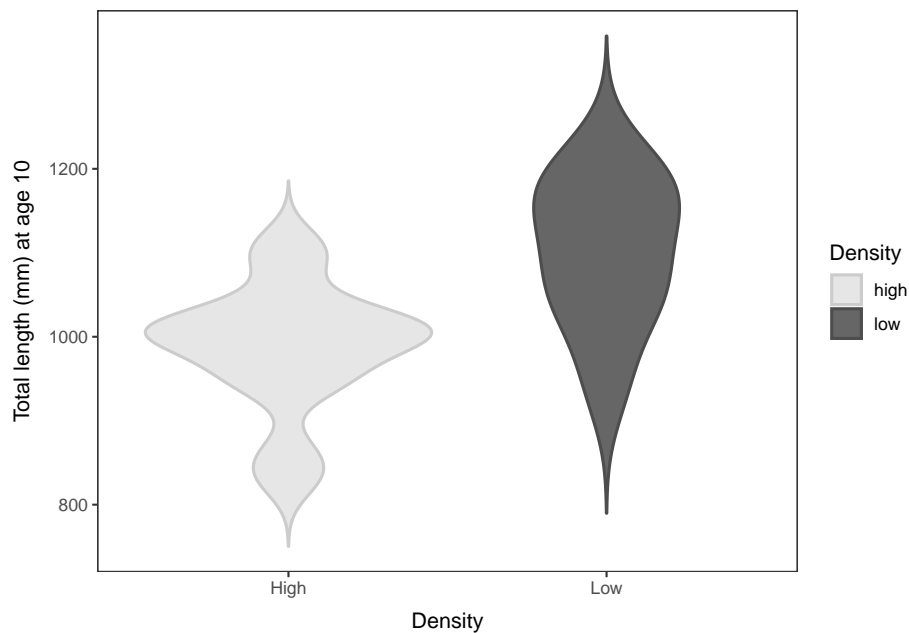


Here is a much better plot. Not that much more difficult to make, and doesn't make you want to puke even if the code does a little bit.

```

mid_carps %>%
  ggplot(aes(x = density, y = Length, fill = density, color = density)) +
    geom_violin(aes(group = density), trim = FALSE, size = .75) +
    scale_x_discrete(breaks=c("high", "low"), labels = c("High", "Low")) +
    scale_fill_grey(start = 0.9, end = 0.4) +
    scale_color_grey(start = 0.8, end = 0.3) +
    xlab("Density") +
    ylab("Total length (mm) at age 10") +
    labs(fill = "Density", color = "Density") +
    theme_bw() +
    theme(
      axis.title.x = element_text(vjust = -1),
      axis.title.y = element_text(vjust = 3),
      panel.grid = element_blank()
    )

```



There, isn't that fancy? I think that gives you a much more detailed understanding of how `Length` varies between `high` and `low` population density than a simple p-value. But, maybe its just me...

6.3 Frequency analysis

Now, what if we didn't collect very good data, or we binned our data into low-resolution categories for the sake of ease in our study design? Often, and for a variety of reasons other than crappy data collection, we want to compare frequencies of events between two (or more) groups. We may even design studies specifically to test these kinds of hypotheses when we think about rates, for example. This is very common in studies of population genetics [definitely citations available for that one - go Google them]

The simplest way to test for differences in the frequency of a categorical response between two groups is (some would argue) the χ^2 test. The χ^2 is another one of those that you should really work out by hand because it is used in a variety of settings "under the hood" of more complex routines. Here is your token video link showing an example. Watch it. Please.

6.3.1 Worked example

Let's say we want to know if the number of grass carp in a given age group (say age 10) varies between years. These fish are sterile hybrids, so we would expect that the number of fish in each age would change drastically with increasing time since the year of initial stocking (1995).

First, make a table showing the number of fish in each **Age** by **Year** with the **grasscarp** data.

```
agexyear <- with(grasscarp, table(Age, Year))

print(agexyear)
```

```
##      Year
## Age 2006 2007 2009 2010 2017
##  1    0    0    0    2    0
##  2    6    7    7    8    0
##  3   11   10   10   14    3
##  4    4    3    6    3    3
##  5    0    1    3    2   10
##  6    0    1    2    1   12
##  7    0    0    1    3   11
##  8    0    1    1    2   12
##  9    1    0    1    3   21
## 10    4    1    1    2   21
## 11    4    9    0    6   15
## 12    3   15    2    3   18
## 13    0    1    4    3    7
```

```
## 14 0 0 11 8 6
## 15 0 0 2 12 12
## 16 0 0 0 2 10
## 17 0 0 0 0 8
## 18 0 0 0 0 9
## 19 0 0 0 0 12
## 20 0 0 0 0 12
## 21 0 0 0 0 5
## 22 0 0 0 0 3
## 23 0 0 0 0 2
```

Basically what we are going to do is analyze the proportion of total fish in each column

You should see some pretty obvious patterns here. We have a couple of things to think about now. First, this is the kind of question you don't need statistics for. Second, we have a whole bunch of empty groups, and these are not random with respect to year. Some of these come from ages that were not yet available in years 2006 - 2010 and some from patterns in fish stocking. The large number of empty pairings and the fact that most age classes had fewer than five fish in any year prior to 2017 means we should probably break the data down a little further. This stinks because we lose resolution, but that is the cost.

For the sake of demonstration, let's summarize the data by **high** and **low** density again and we'll look at the number of fish collected in each age class during high and low density years.

```
freqs <- grasscarp %>% # Pass grass carp data frame to group_by()
  filter(Age %in% c(10:15)) # Select only shared age range

head(freqs)
```

```
##   Year Length Weight_kg Age cohort n_stocked      n acre   ha nha      kg kg_ha
## 1 2006  1262   23.154  12  1994       7000 25128 2957 1203  21 129929  108
## 2 2006  1138   23.923  10  1996       7000 25128 2957 1203  21 129929  108
## 3 2006  1187   18.614  10  1996       7000 25128 2957 1203  21 129929  108
## 4 2006  1207   19.522  12  1994       7000 25128 2957 1203  21 129929  108
## 5 2006  1086   21.285  10  1996       7000 25128 2957 1203  21 129929  108
## 6 2006  1095   18.160  12  1994       7000 25128 2957 1203  21 129929  108
##   density
## 1      low
## 2      low
## 3      low
## 4      low
## 5      low
## 6      low
```


We will test the null hypothesis that there is no difference in the number of age-10 fish between high and low densities.

```
# Run the test
chi_test <- with(freqs, chisq.test(x = density, y = Age))

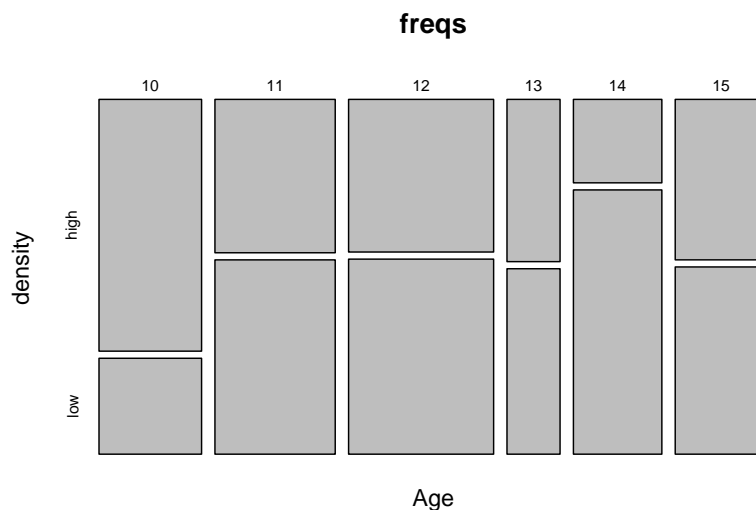
# Have a look
print(chi_test)
```

```
##
## Pearson's Chi-squared test
##
## data: density and Age
## X-squared = 13.107, df = 5, p-value = 0.0224
```

And, bam! We see that there is a difference in the frequency of fish collected in each age class in high and low density years. Shocker.

Data visualization techniques for contingency table analyses like this seem to have generally lagged behind theory in terms of wide-spread implementation. There is a base R `mosaicplot` that plots relative frequencies. You can interpret the width of the bars as the proportion of total observations in each age class. Likewise, the height of the vertical segments corresponds to proportion of **high** or **low** density observations in each **Age**.

```
mosaicplot(Age ~ density, data = freqs)
```



The need for improved graphical representation for these data types is recognized. There have been recent efforts to extend the philosophies used `ggplot()` to contingency analysis by `r` developers (see Wickham and Hofmann 2011<>). It was even the topic of a recent master's thesis (see Grant 2017). But as far as I know the ideas from these works have not been integrated into `ggplot2` or the `tidyverse` yet. Sorry about the citations. I don't know what I was thinking. This is supposed to be the Worst Stats Text ever.

6.4 Next steps

In this chapter, we introduced inferential statistics and walked through examples of a few simple statistical tests for comparing samples to one another using two-sample tests, or to a single value using a one-sample test. In Chapter 7 we will continue to build on these tools as we press on to linear models and the rest of statistics.