R for Ecological Data Science: A Gentle Introduction

Nathan L. Brouwer

2018-09-13

Contents

Preface

This book grew out of exercises used for the computer lab portion of ENS 495:Design & Analysis, an ecological statistics course I taught for two years as and adjunct with the Department of Department of Biological & Environmental Sciences at California University of Pennsylvania. The goal is to walk students step by step through the process of working with data in R.

A major influence in how this book is written is th online book Program MARK: a gentle introduction, edited Evan Cooch and Gary White. *MARK* is one of the most important pieces of software ever written for ecologists and implements a vast number of mark-recapture and occupancy-type models for understanding population processes such as population size, survival, migration, occupancy, and meta-population dynamics. The book patiently walks users through the basic of these models step by step, concept by concept, and click by click. I have sought to adopt a similar thorough and holistic approach in this book.

Part I

Introduction

In Part I of this book I'll introduce why the role of data science in ecology is important and lay out the scope of this book. I'll also walk through the basic steps of getting R and RStudio up and running on your comptuer, walk through a basic R section, and discuss the various ways we'll interact with R using **code** and **script files**. I'll also introduce **rmarkdown**, a way of integrating the R code of scripts with basic capabilities of word processing and web developement. Scripts organized in rmarkdown are a handy way to organize and **annotate code**, store and disseminate your work, and structure your analyses and data presensations so that the computations behind them are fully **reproducible**.

Chapter 1

What is Ecological Data Science & How Will this Book Teach It?

[**NOTE: This section is currently under development. The paper by Touchon & McCoy (2016) and its references lay out many of the reasons for the statistical focus of this book.**]

> "Ecological questions and data are becoming increasingly complex and as a result we are seeing the development and proliferation of sophisticated statistical approaches in the ecological literature. … It is no longer sufficient to only ask 'whether' or 'which' experimental manipulations significantly deviate from null expectations. Instead, we are moving toward parameter estimation and asking **'how much'** and in **'what direction'** ecological processes are affected by different mechanisms" (Touchon & McCoy 2016, Ecosphere, emphsis mine)

> "Spreadsheets are often used as the basis of data collection and education; but this is potentially problematic since spreadsheets typically do not promote good data management practices…. The features of spreadsheets that make them desirable for the average researcher, such as extensibility, use of formatting for organization, embedding charts, make them undesirable for preparing data for long-term archiving and reuse."(Strasser & Hampton 2012 Ecosphere)

1.1 What is data science?

> Data analysis include "procedures for analyzing data, techniques for interpreting the results…, ways of planning the gathering of data to make its analysis easier, more precise or more accurate, and all the machinery and results of (mathematical) statistics which apply to analyzing data." John Tukey, "The future of data analysis", Annals of Mathematical Statistics, 1962.

- People argue about what data science is
- What Tukey calls "data analysis" is now termed "data science" by many.

- Some define data science as closely allied with computer science and want its use most closely associated with things like "big data", data mining, machine learning, and artificial intelligence.
- Others, such as RStudio's Hadley Whickham (creator of ggplot2, dplyr, and most of the infrasture of the tidyverse of R package) define it more broadly to involve all aspects of the life cycle of data.
- (Wickham also defines a data scientists as "A data scientist is a statistician who is wearing a bow tie" https://twitter.com/hadleywickham/status/906146116412039169?lang=en)

1.2 Goals

"The rise of computer programming, computational power, and modern statistical approaches may…" allow "…scientists to ask new questions and to extract more information from data than ever before." (Touchon & McCoy 2016, Ecosphere)

- **Statistical computing** using R, RStudio, and rmarkdown
- **Data analysis**, from t-tests to mixed models in R
- **Current statistical practice**, with an emphasis on **statistical modeling** and **effect size estimation** instead of "statistical tests"
- **Data visualization**, with an emphasis on ggplot2
- **Data science**, from data management best practices to data cleaning with dplyr
- **Computational reproducibility**, from formatting scripts to using rmarkdown to write reproducible reports
- Dropping things that aren't needed, like classical rank-based nonparametric methods.

1.3 Approach

- always explore and visualize data
- step-by-step instructions
- frequently refreshing and review
- comprehensive and self-contained

1.4 Requirements

- R
- RStudio
- External packages loaded via RStudio

1.5 Refereces

Strasser & Hampton 2012. The fractured lab notebook: undergraduates and ecological data management training in the United States. EcoSphere. https://esajournals.onlinelibrary.wiley.com/doi/abs/10.1890/ES12-00139.1

Touchon & McCoy 2017. The mismatch between current statistical practice and doctoral training in ecology. EcoSphere. https://esajournals.onlinelibrary.wiley.com/doi/abs/10.1002/ecs2.1394

Tukey. 1962. The future of data analysis. Annals of Mathematical Statistics. https://www.jstor.org/stable/2237638

1.6 Bibliography

Relevant papers cited by Touchon & McOy 2017.

Barraquand, F., T. H. G. Ezard, P. S. Jørgensen, N. Zimmerman, S. Chamberlain, R. Salguero-Gómez, T. J. Curran, and T. Poisot. 2014. Lack of quantitative training among early-career ecologists: a survey of the problem and potential solutions. PeerJ 2:e285.

Butcher, J. A., J. E. Groce, C. M. Lituma, M. C. Cocimano, Y. Sánchez-Johnson, A. J. Campomizzi, T. L. Pope, K. S. Reyna, and A. C. S. Knipps. 2007. Persistent controversy in statistical approaches in wildlife sciences: a perspective of students. Journal of Wildlife Management 71:2142–2144

Ellison, A. M., and B. Dennis. 2009. Paths to statistical fluency for ecologists. Frontiers in Ecology and the Environment 8:362–370.

Germano, J. D. 2000. Ecology, statistics, and the art of misdiagnosis: the need for a paradigm shift. Environmental Reviews 7:167–190.

Quinn, J. F., and A. E. Dunham. 1983. On hypothesis testing in ecology and evolution. American Naturalist 122:602–617.

Chapter 2

What is R and why use it?

[these notes are from a lecture and have not been re-written much yet]

R is a powerful piece of software used for data science and data analysis. In this chapter I will briefly introduce the advantages of using R, why you might want to learn it, and also indicate some alternatives and adjuncts you could consider.

2.1 How do we typically use software in science?

Most scientists rely on both general and specialized pieces of software for various parts of their work. For data entry they likely use spreadsheet software Excel, though increasingly Google Sheets. For data analysis they might use one of many options, such as GraphPad Prism, Minitab, SAS, SPSS, or STATA. For making plots, many people will export their export their results back to Excel, while others use specialized software like SigmaPlot. Many scientists also use specialized programs; in ecology many researchers do GIS in ArcGIS or QGIS, mark-recapture analysis in Program MARK or Distance, use RAMAS or Vortex for population viability analysis, or build custom mathematical programs in MatLab or Python. If they do multivariate statistics like [ordination](https://en.wikipedia.org/wiki/Ordination_(statistics) the may use a specialized stats program like PC-ORD.

Since software can be expensive, some scientists will rely on Excel for all of their work. Excel can do many things, but it can't do everything all the specialized types of software can do. Moreover, its very limited in the range of statistics it can do and graphs it can make.

2.2 What does R do?

R is amazing because it has been explicitly developed to do several things very well, particularly statistics, math, making great-looking figures, and writing computer programs to automate these tasks. Additionally, R has been extended by developers to be able to be a powerful tool for data cleaning and organization, to be used as a GIS, and as an integrated word processor and website make for publishing work.

2.3 Why use R

In addition to is many capabilities, R has the advantage this it is

- free anyone, always

- used by statisticians to develop new statistical techniques, so new techniques often come out 1st in R
- used by almost all ecological statisticians to develop new techniques (mark recapture, distance sampling)

2.4 Who uses it?

R continues to increase in popularity. Among data scientists it is second only to Python. Among academics it has eclipsed SAS in many fields. It is also used by analyses in many large companies, such Facebook, and by journalists looking for stories in or reporting on large volumes of data

see http://blog.revolutionanalytics.com/2014/05/companies-using-r-in-2014.html for further discussion.

2.5 R and computational reproducibility

One factor potentially contributing to R's popularity, or at least a major bonus for using it, is ease of use for making analyses reproducible. All commands in R are typed out and the best way to do this is in a static **script file** from which you send commands to R to execute. This creates a record of your analyses. This feature is shared by other programs such as SAS and Stats, and other programming languages such as Matlab and Python. The advantage of R is that the script files are simply plain text files which anyone can open and - if they've downloaded R, which is free - they can run. Developers have also created numerous tools for creating **reproducible analysis workflows** and which allow R to be used in all data-related aspects of a project, from **data cleaning** to **formatting journal submissions**. What this means is that without become an expert programmer you can set up your work so that you can re-run all of your data cleaning, analyses, and graph building with a single command in R. This makes what you've done auditable, transparent, and easy to re-use for future work.

2.6 Alternatives to R

R has many advantages, but it has one critical issue: the learning curve. R is a command-line driven analysis tool, which means you type out specific commands for almost everything single thing R does. Excel is pretty user friendly, and several stats programs similarly use point-and-click interfaces, such as SPSS, JMP, and Stata SAS also requires a lot of command writing, but is generally consider more user friendly than R.

Recently, two free point-and-click statistical analysis programs have been release that are built on R but require no programming. JASP ("Just another statistics program") has an emphasis on Bayesian statistics, particularly Bayesian hypothesis testing using Bayes factors (an approach increasing in popularity, especially in psychology, but which some Bayesians, like Andrew Gelman, disavow). While JASP is based on R, it does not currently allow access to the underlying R code.

Jamovi has a similar spirit as JASP (indeed, it was founded by developers who had worked on JASP) but is more transparent about the underlying R code being used to run the analysis.

Chapter 3

A first encounter with R & RStudio

Nathan Brouwer, Phd brouwern at gmail.com https://github.com/brouwern Twitter lobrowR

Vocabulary

- console
- script editor / source viewer
- interactive programming
- scripts / script files
- .R files
- text files / plain text files
- command execution / execute a command from script editor
- comments / code comments
- commenting out / commenting out code
- stackoverflow.com
- the rstats hashtag

R commands

- c(…)
- mean(…)
- sd(…)
- ?
- read.csv(…)

3.1 Getting Started With R and RStudio

- R is a piece of software that does calculations and makes graphs.
- RStudio is a GUI (graphical user interface) that acts as a front-end to R
- Your can use R directly, but most people use a GUI of some kind
- RStudio has become the most popular GUI

The following instructions will lead you click by click through downloading R and RStudio and starting an initial session. If you have trouble with downloading either program go to YouTube and search for something like "Downloading R" or "Installing RStudio" and you should be able to find something helpful, such as "How to Download R for Windows".

3.2 Getting R onto your computer

To get R on to your computer first go to the CRAN website at https://cran.r-project.org/ (CRAN stands for "comprehensive R Archive Network"). At the top of the screen are three bullet points; select the appropriate one (or click the link below)

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

Each page is formatted slightly differently. For a current Mac, click on the top link, which as of 8/16/2018 was "R-3.5.1.pkg" or click this link. If you have an older Mac you might have to scroll down to find your operating system under "Binaries for legacy OS X systems."

For PC select "base" or click this link.

When its downloaded, run the installer and accept the default.

3.3 Getting RStudio on to your computer

RStudio is an R interface developed by a company of the same name. RStudio has a number of commercial products, but much of their portfolio is freeware. You can download RStudio from their website www.rstudio.com/ . The download page (www.rstudio.com/products/rstudio/download/) is a bit busy because it shows all of their commercial products; the free version is on the far left side of the table of products. Click on the big green DOWNLOAD button under the column on the left that says "RStudio Desktop Open Source License" (or click on this link).

This will scroll you down to a list of downloads titled "Installers for Supported Platforms." Windows users can select the top option RStudio 1.1.456 - Windows Vista/7/8/10 and Mac the second option RStudio 1.1.456 - Mac OS X 10.6+ (64-bit). (Versions names are current of 8/16/2018).

Run the installer after it downloads and accept the default. RStudio will automatically link up with the most current version of R you have on your computer. Find the RStudio icon on your desktop or search for "RStudio" from your task bar and you'll be read to go.

3.3.1 Keep R and RStudio current

Both R and RStudio undergo regular updates and you will occasionally have to re-download and install one or both of them. In practice I probably do this about every 6 months.

3.4 Getting started with R itself (or not)

This is a walk-through of a very basic R session. It assumes you have successfully installed R and RStudio onto your computer, and nothing else.

Most people who use R do not actually use the program itself - they use a GUI (graphical user interface) "front end" that make R a bit easier to use. However, you will probably run into the icon for the underlying R program on your desktop or elsewhere on your computer. It usually looks like this:

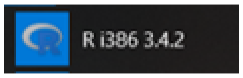


Figure 3.1: The R logo

Figure 3.2: The RStudio logo
(#fig:rstudio.logo)

The long string of numbers have to do with the version and whether is 32 or 64 bit (not important for what we do).

If you are curious you can open it up and take a look - it actually looks a lot like RStudio, where we will do all our work (or rather, RStudio looks like R). Sometimes when people are getting started with R they will accidentally open R instead of RStudio; if things don't seem to look or be working the way you think they should, you might be in R, not RStudio

3.5 Getting started with RStudio

Now we'll get started with RStudio. We'll get to know what it looks like and configure it a bit for out needs.

3.5.1 RStudio at first glance

The RStudio logo looks like this.

When you open up you'll be greeted by a fairly busy array of menus and things. Don't panic! A typical fresh starting point in RStudio is shown in Figure 2.

When referring to RStudio, there are two terms that need to be understood. As shown in Figure 3, there is the **console** section of RStudio and the **script editor** or **source viewer**.

A "cheat sheet" called the "RStudio IDE Cheat Sheet" details all of RStudio's many features and is available at https://www.rstudio.com/resources/cheatsheets/ . It very thorough, though a bit dense.

3.5.2 The console versus the script editor

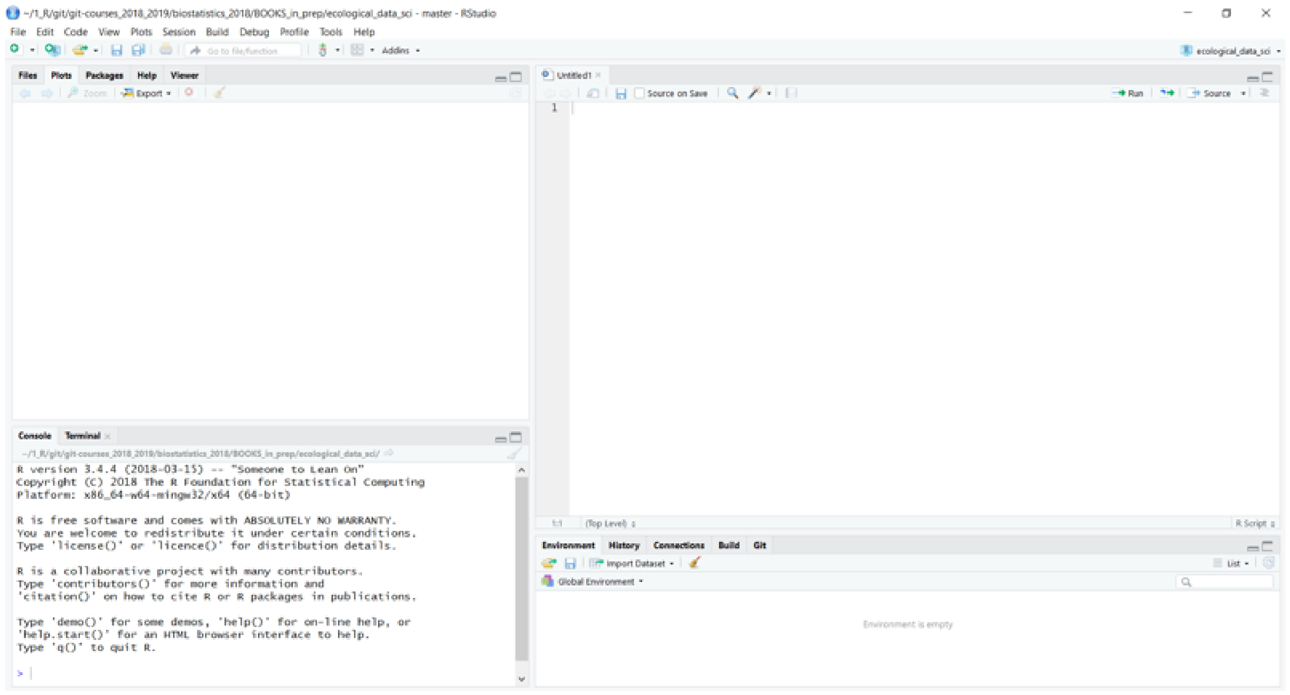
You can type and enter text into both the console and the script editor. The console, however, respond like a calculator, while the script editor works more like a text editor.

3.5.2.1 The R console

The **console** in RStudio act exactly the same way as it does in R. This is an **interactive programming** situation that is very similar to a scientific calculator. If you click your mouse inside the console and type "1 + 1" then press enter you will see the following type of output

```
1 + 1
```

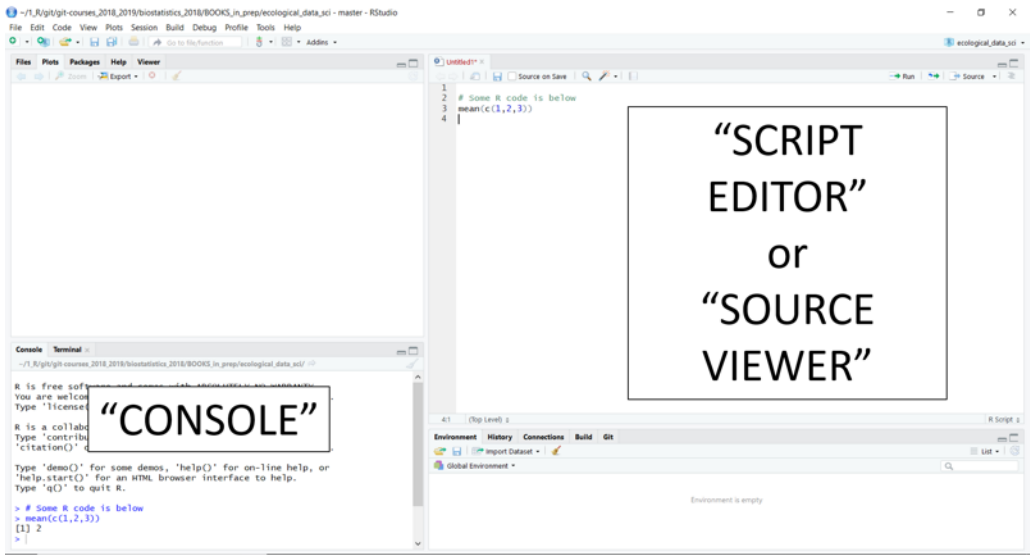
```
## [1] 2
```


~/1_R/git/git-courses_2018_2019/biostatistics_2018/BOOKS_in_prep/ecological_data_sci - master - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
ecological_data_sci
Files Plots Packages Help Viewer
Zoom
Export
Untitled1
Source on Save
Run
Source
Console Terminal
~/1_R/git/git-courses_2018_2019/biostatistics_2018/BOOKS_in_prep/ecological_data_sci/
1:1
(Top Level)
R Script
Environment History Connections Build Git
Import Dataset
List
Global Environment
Environment is empty

```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

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

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

>
```

Figure 3.3: RStudio when first opened.

(#fig:rstudio.open)


~/1_R/git/git-courses_2018_2019/biostatistics_2018/BOOKS_in_prep/ecological_data_sci - master - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
ecological_data_sci
Files Plots Packages Help Viewer
Zoom
Export
Untitled1*
Source on Save
Run
Source
“SCRIPT EDITOR” or “SOURCE VIEWER”
Console Terminal
~/1_R/git/git-courses_2018_2019/biostatistics_2018/BOOKS_in_prep/ecological_data_sci/
“CONSOLE”
4:1
(Top Level)
R Script
Environment History Connections Build Git
Import Dataset
List
Global Environment
Environment is empty

```

# Some R code is below
mean(c(1,2,3))

```

```
R is free soft
You are welcon
Type 'license

R is a collabo
Type 'contribu
'citation()' o

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

> # Some R code is below
> mean(c(1,2,3))
[1] 2
>
```

Figure 3.4: RStudio’s console and script editor.

(#fig:rstudio.console)

Note that right in front of where you typed "1+1" there is a ">" symbol. This is always in the R console and never needs to be typed.

One thing to note about R is that it's not particular about spacing. All of the following things will yield the same results

```
1+1
1 + 1
1          +         1
```

3.5.2.2 R's console as a scientific calculator

You can interact with R's console similar to a scientific calculator. For example, you can use parentheses to set up mathematical statements like

```
5*(1+1)
```

```
## [1] 10
```

Note however that you have to be explicit about multiplication. If you try the following it won't work.

```
5(1+1)
```

R also has built-in functions that work similar to what you might have used in Excel. For example, in Excel you can calculate the average of a set of numbers by typing "=average(1,2,3)" into a cell. R can do the same thing except

- The command is "mean"
- You don't start with "="
- You have to package up the numbers like what is shown below using "c(…)"

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

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

Where "c(…)" packages up the numbers the way the mean() function wants to see them.

If you just do the following R will give you an answer, but its the wrong one

```
mean(1,2,3)
```

This is a common issue with R – and many programs, really – it won't always tell you when somethind didn't go as planned. This is because it doesn't know something didn't go as planned; you have to learn the rules R plays by.

3.5.2.3 Practice: math in the console

See if you can reproduce the following results

Division

```
10/3
```

```
## [1] 3.333333
```

The standard deviation

```
sd(c(5,10,15)) # note the use of "c(...)"
```

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

3.5.2.4 The script editor

While you can interact with R directly within the console, the standard way to work in R is to write what are known as **scripts.** These are computer code instructions written to R in a **script file.** These are save with the extension **.R** but area really just a form of **plain text file.**

To work with scripts, what you do is type commands in the script editor, then tell R to **excute** the command. This can be done several ways.

First, you tell RStudio the line of code you want to run by either * Placing the cursor at the end a line of code, OR * Clicking and dragging over the code you want to run in order highlight it.

Second, you tell RStudio to run the code by * Clicking the "Run" icon in the upper right hand side of the script editor (a grey box with a green error emerging from it) * pressing the control key ("ctrl)" and then then enter key on the keyboard

The code you've chosen to run will be sent by RStudio from the script editor over to the console. The console will show you both the code and then the output.

You can run several lines of code if you want; the console will run a line, print the output, and then run the next line. First I'll use the command mean(), and then the command sd() for the standard deviation:

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

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

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

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

3.5.2.5 Comments

One of the reasons we use script files is that we can combine R code with **comments** that tell us what the R code is doing. Comments are preceded by the hashtag symbol #. Frequently we'll write code like this:

```
#The mean of 3 numbers
mean(c(1,2,3))
```

If you highlight all of this code (including the comment) and then click on "run", you'll see that RStudio sends all of the code over console.

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

Comments can also be placed at the *end* of a line of code

```
mean(c(1,2,3)) #Note  the use of c(...)
```

Sometimes we write code and then don't want R to run it. We can prevent R from executing the code even if its sent to the console by putting a "#" *infront* of the code.

If I run this code, I will get just the mean but not the sd.

```
mean(c(1,2,3))
#sd(c(1,2,3))
```

Doing this is called **commenting out** a line of code.

3.6 Help!

There are many resource for figuring out R and RStudio, including

- R's built in "help" function
- Q&A websites like **stackoverflow.com**
- twitter, using the hashtag #rstats
- blogs
- online books and course materials

3.6.1 Getting "help" from R

If you are using a function in R you can get info about how it works like this

```
?mean
```

In RStudio the help screen should appear, probably above your console. If you start reading this help file, though, you don't have to go far until you start seeing lots of R lingo, like "S3 method","na.rm", "vectors". Unfortunately, the R help files are usually not written for beginners, and reading help files is a skill you have to acquire.

For example, when we load data into R in subsequent lessons we will use a function called "read.csv"

Access the help file by typing "?read.csv" into the console and pressing enter. Surprisingly, the function that R give you the help file isn't what you asked for, but is read.table(). This is a related function to read.csv, but when you're a beginner thing like this can really throw you off.

Kieran Healy as produced a great cheatsheet for reading R's help pages as part of his forthcoming book. It should be available online at http://socviz.co/appendix.html#a-little-more-about-r

3.6.2 Getting help from the internet

The best way to get help for any topic is to just do an internet search like this: "R read.csv". Usually the first thing on the results list will be the R help file, but the second or third will be a blog post or something else where a usually helpful person has discussed how that function works.

Sometimes for very basic R commands like this might not always be productive but its always work a try. For but things related to stats, plotting, and programming there is frequently lots of information. Also try searching YouTube.

3.6.3 Getting help from online forums

Often when you do an internet search for an R topic you'll see results from the website www.stackoverflow.com, or maybe www.crossvalidated.com if its a statistics topic. These are excellent resources and many questions that you may have already have answers on them. Stackoverflow has an internal search function and also suggests potentially relevant posts.

Before posting to one of these sites yourself, however, do some research; there is a particular type and format of question that is most likely to get a useful response. Sadly, people new to the site often get "flamed" by impatient pros.

3.6.4 Getting help from twitter

Twitter is a surprisingly good place to get information or to find other people knew to R. Its often most useful to ask people for learning resources or general reference, but you can also post direct questions and see if anyone responds, though usually its more advanced users who engage in twitter-based code discussion.

A standard tweet might be "Hey #rstats twitter, am knew to #rstats and really stuck on some of the basics. Any suggestions for good resources for someone starting from scratch?"

3.7 Other features of RStudio

3.7.1 Ajusting pane the layout

You can adjust the location of each of RStudio 4 window panes, as well as their size.

To set the pane layout go to 1. ”Tools” on the top menu 1. ”Global options” 1. “Pane Layout”

Use the drop-down menus to set things up. I recommend 1. Lower left: “Console”” 1. Top right: “Source” 1. Top left: “Plot, Packages, Help Viewer” 1. This will leave the “Environment…” panel in the lower right.

3.7.2 Adjusting size of windows

You can clicked on the edge of a pane and adjust its size. For most R work we want the console to be big. For beginners, the “Environment, history, files” panel can be made really small.

3.8 Practice (OPTIONAL)

Practice the following operations. Type the directly into the console and execute them. Also write them in a script in the script editor and run them.

Square roots

```
sqrt(42)
```

```
## [1] 6.480741
```

The date Some functions in R can be executed within nothing in the parentheses.

```
date()
```

```
## [1] "Thu Sep 13 00:15:35 2018"
```

Exponents The ^ is used for exponents

```
42^2
```

```
## [1] 1764
```

A series of numbers A colon between two numbers creates a series of numbers.

```
1:42
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
```

logs The default for the log() function is the natural log.

```
log(42)
```

```
## [1] 3.73767
```

log10() gives the base-10 log.

```
log10(42)
```

```
## [1] 1.623249
```

exp() raises e to a power

```
exp(3.73767)
```

```
## [1] 42.00002
```

Multiple commands can be nested

```
sqrt(42)^2
log(sqrt(42)^2)
exp(log(sqrt(42)^2))
```

Chapter 4

The different faces of R code: The console, scripts & RMarkdown

There are a number of ways to interact with R

- Directly in the **console**, like a scientific calculator
- Using **script files** (.R) with
- R code types out and sent over to the console
- notes "commented out" using hastags "#"
- **rmarkdown** (.Rmd) files with
- R code in species **code chunks**
- notes written like in a word precessor
- formatting using **markdown**, a **markup language**

This chapter will briefly introduce these different ways of working in R

4.1 The console

In a previous chapter we introduced the **console.** You can interact with the console in a similar manner as a scientific calculator.

As you execute more commands, the move up the screen. You can scroll back up to see what you've done previously.

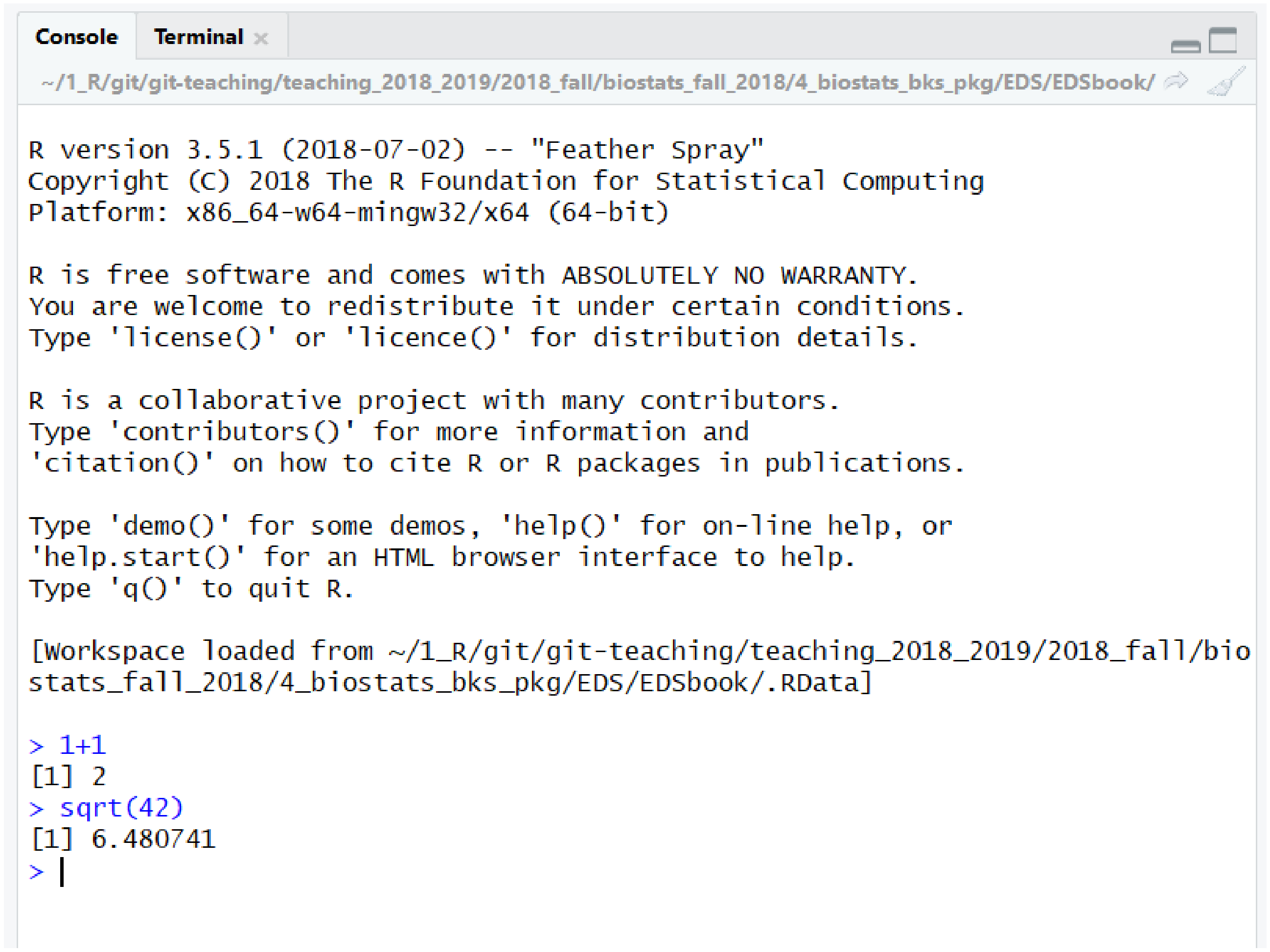
4.2 Scripts

You can create a record of the commands you have executed in the console, but this isn't a very efficient way to work. If you want to keep track of the commands you're running (and often you do) its best to write them in a script file and then send them over to the console to execute the code.

4.2.1 Creating scripts

When you open RStudio a blank script file will be open. Subsequently, RStudio will open files you have worked with previously. To create a new blank script file:

- Click on File


Console
Terminal
~/1_R/git/git-teaching/teaching_2018_2019/2018_fall/biostats_fall_2018/4_biostats_bks_pkg/EDS/EDSbook/

```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

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

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

[Workspace loaded from ~/1_R/git/git-teaching/teaching_2018_2019/2018_fall/bio
stats_fall_2018/4_biostats_bks_pkg/EDS/EDSbook/.RData]

> 1+1
[1] 2
> sqrt(42)
[1] 6.480741
> 
```

Figure 4.1: The RStudio console

(#fig:first.chnk.sxn1ch4)

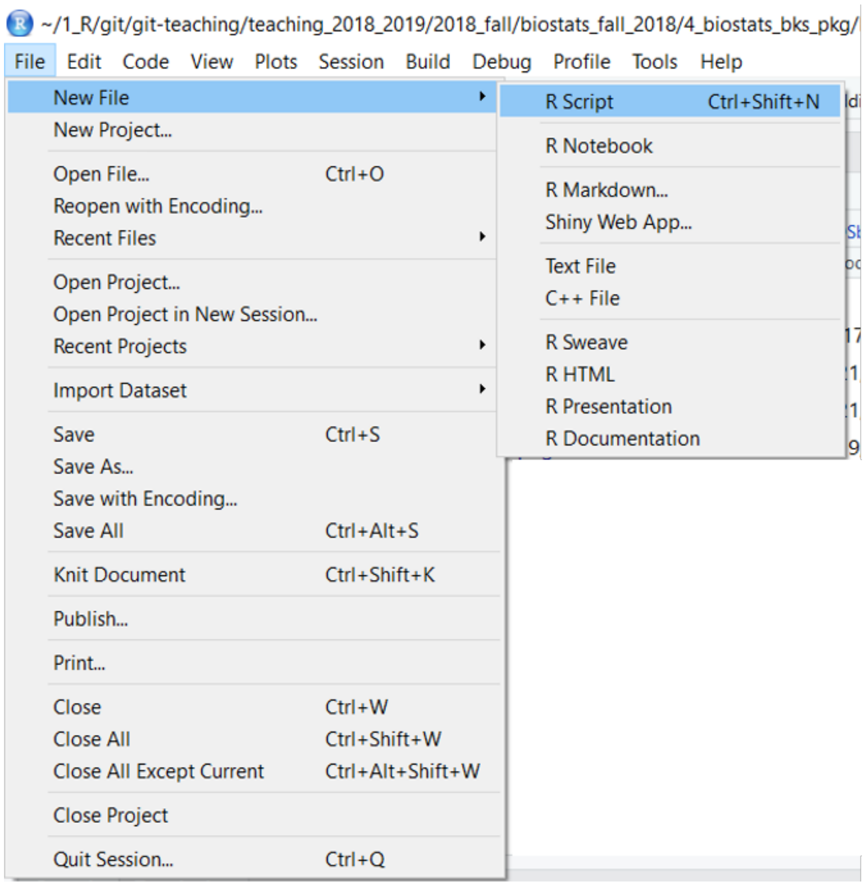


Figure 4.2: Creating a new R script file

- New File
- R script

or just type control + shift + N (on a PC) or command + shift + N on (?on mac), which is similar how you make a new document in most programs.

You can create or open multiple scripts, which RStudio organizes as tabs like in a web browser.

4.2.2 Running code from a script

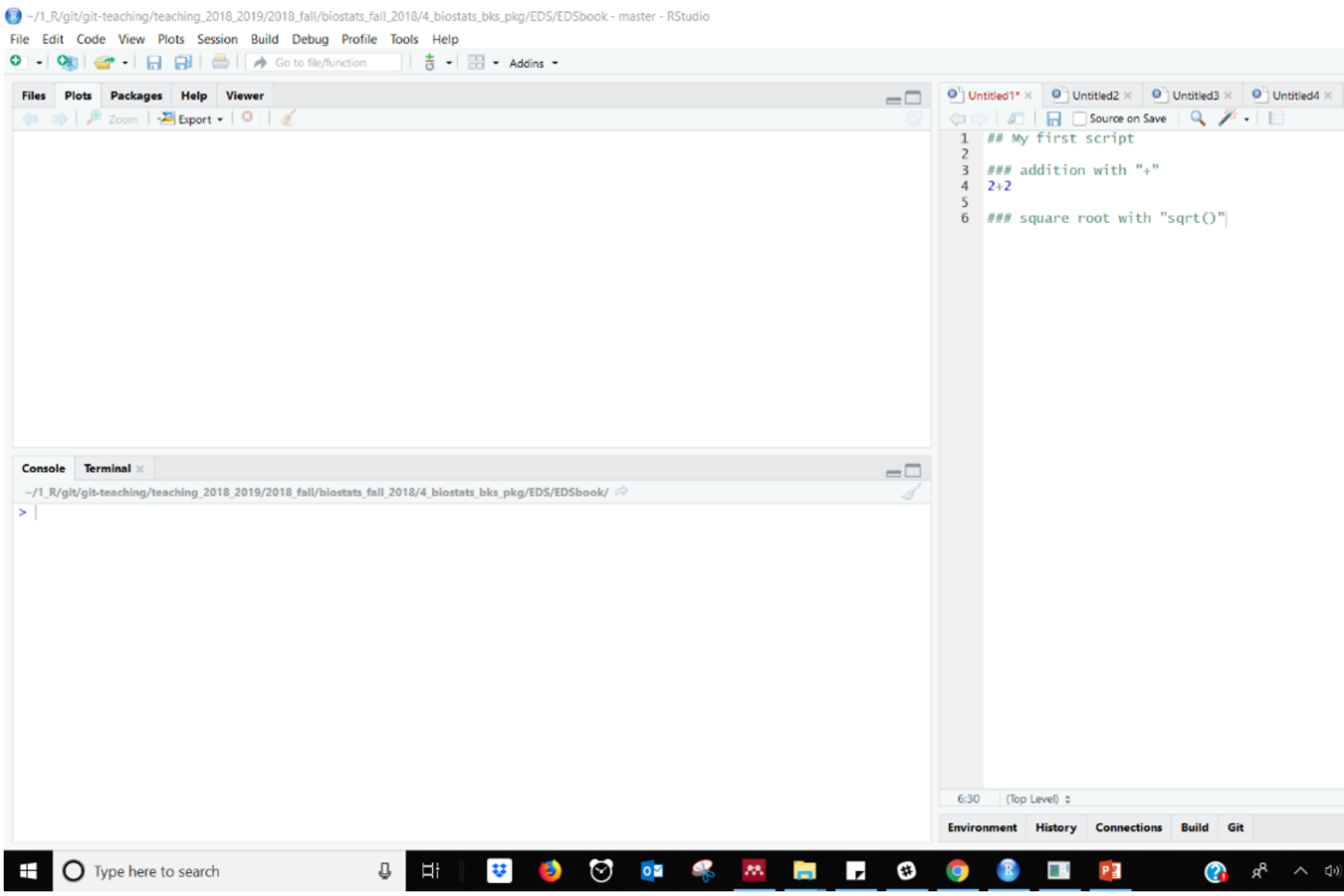
To run a code you can either place click to the righ of the line of code and click the "run" button.

You can also highlight the code by clicking and dragging over it. This is useful when you have multiple lines of code.

4.2.3 Running code with keyboard shortcuts

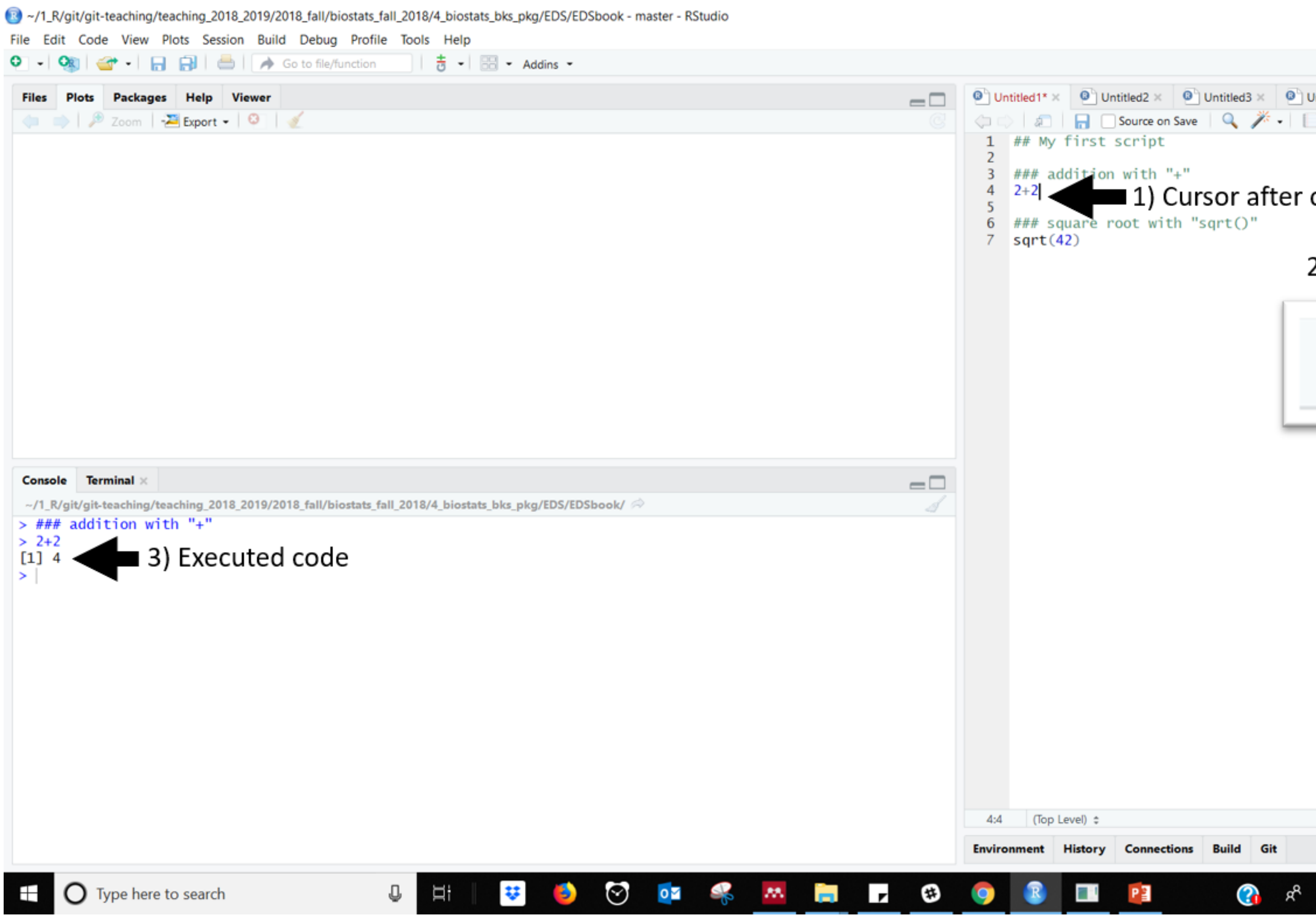
Want to look like an R pro? Learn keyboard shortcuts so you don't have to use the mouse. Both of the above methods work using simple keyboard shortcuts:

- PCs: Control + Enter
- Macs: Command + Enter (?)


~/1_R/git/git-teaching/teaching_2018_2019/2018_fall/biostats_fall_2018/4_biostats_bks_pkg/EDS/EDSbook - master - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Files Plots Packages Help Viewer
Console Terminal
Untitled1* Untitled2 Untitled3 Untitled4
Source on Save
Environment History Connections Build Git

```
## My first script

### addition with "+"
2+2

### square root with "sqrt()"
```

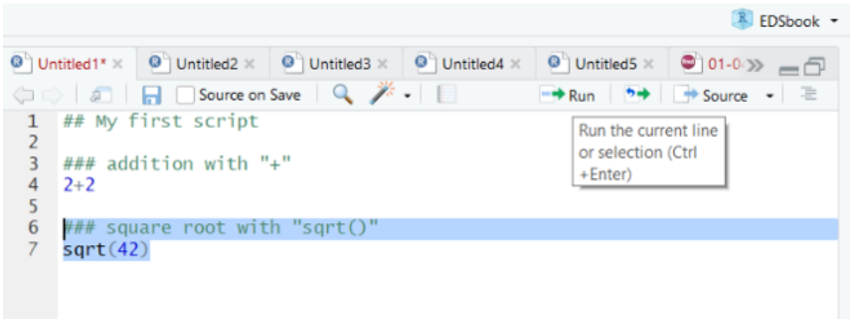
Figure 4.3: R Script file with comments in RStudio


~/1_R/git/git-teaching/teaching_2018_2019/2018_fall/biostats_fall_2018/4_biostats_bks_pkg/EDS/EDSbook - master - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Files Plots Packages Help Viewer
Console Terminal
Untitled1* Untitled2 Untitled3
Source on Save
1) Cursor after c
3) Executed code
Environment History Connections Build Git
Type here to search

```
## My first script

### addition with "+"
2+2

### square root with "sqrt()"
sqrt(42)
```

```
> ### addition with "+"
> 2+2
[1] 4
>
```

Figure 4.4: Running code using RStudios RUN button. Note cursor to the right of the code


EDSbook
Untitled1* Untitled2 Untitled3 Untitled4 Untitled5 01-0
Source on Save
Run
Source
Run the current line or selection (Ctrl +Enter)

```
## My first script

### addition with "+"
2+2

### square root with "sqrt()"
sqrt(42)
```

Figure 4.5: Highlighted code in an RStudio script

Another handy shortcut is "Control + 2", which moves your computer's cursor from the script editor to the console. (Control + 2 moves it from console to editor)

4.3 Organizing scripts

> "You mostly collaborate with yourself, and me-from-two-months-ago never responds to email." (Karen Cranston, paraphrasing Mark Holder; quoted by Megan Duffy on dynamicecology)

Script files perform a record of your work so you can

- remember what you did
- re-run it to check things
- re-use your code for new analyses
- track down errors (which will happen!)
- share with collaborators

Script files are not unique to R, but the R community seems to have built up a particularly good infrasture for their implemenation and ethos encouraging their use. Megan Duffey at Dynamic Ecology has an excellent post on this.

When you first start out learning R most of your scripts will be disposable. Quickly you'll want to start keeping track of the code you write in class to refer back to. When you start doing analyses you'll want to write comments as you go, and provide details at the top of your file so you can quickly get up to speed when you come back to the file.

4.3.1 What to include in a script

A good R script should be a self-sufficient document that your future self can easily make sense of, or better yet, someone starting from scratch can understand. Depending on the exact purpose, things to include might be

- A general title, such as "R Script: data exploration &t-test for analaysis of frog arm girth"
- Who wrote it and their contact info
- When the script was created
- when it was most recently accessed or created
- What data it uses and where it comes from
- What project or paper it relates to

A challenge when writing and maintaining R scripts is that you are often actively engaged in learnign R, learning stats, and learning about or exploring your data. So you write a lot of code then erase it, or scratch out code in a script and then move on. While I have written and saved many scripts that I have never re-opened, I have never re-opened a script and said "wow, I went WAY overboard annotating this thing!" ALso, commenting code makes it much easier to read; I often add fairly simple comments to make code easier navigate and to break things up into smaller chunks.

So, at a minimum I think every script should

- Have some kind of header saying what it is and when it was made
- Have one line of comments or annotations for every 3 to 5 lines of code.

4.3.2 Formatting sections in R scripts

To make scripts easier to navigate its useful to strong together the comment character "#" to make dividers and boxes. This is very good practice to make code more readable. It can be a bit tedious at times to do this; one advantage of rmarkdown, which we'll introduce at the end of this chapter and go into further in the next, is that it makes it very easy to format section titles.

4.3.3 A sample R script

On the following pages are examples of R scripts for a formal analysis of a dataset. First I'll show what the script might look like as I write it. Then I'll show how I'll fix it up once I know its something I am going to look back at in the fugure

```
### R Script: Analysis of frog arm girth

## Nathan Brouwer (brouwern@gmail.com)
## 6/6/2018
## update: 8/17/2018

## I am re-running analysis from paper by Buzatto  et al 2015
## I want to compare the results of a t-test with
## and w/o Welche's correction for unequal variation

## Packages
library(wildlifeR)

## Data set up
# load frogarm data from Buzatto et al 2015
data("frogarms")


## Data visualization
# histogram of all data
hist(frogarms$sv.length)
```

```
## Data analysis
# unpaired t-test NOT using Welch's correction
## NOTE: assumes variation within each group EQUAL
t.test(sv.length ~ sex,    # model formula
       var.equal = TRUE,   # set variances to equal
         data = frogarms) # data

# unpaired t-test >>using<< Welch's correctin
t.test(sv.length ~ sex,
       var.equal = FALSE, # set variances to be unequal
         data = frogarms)
```

4.3.4 A polished R script

```
###########################################
###
### R Script: Analysis of frog arm girth
###
###########################################

## Author:      Nathan Brouwer (brouwern@gmail.com)
## Creation:    6/6/2018
## Last update: 8/17/2018


###############
## Introduction
###############

# This script is an analysis of frog body size and arm girth

## I am re-running analysis from paper by Buzatto  et al 2015
## I want to compare the results of a t-test with
## and w/o Welche's correction for unequal variation
```

```
# Data were originally from
# Buzatto  et al 2015. Sperm competition and the evolution of
#       precopulatory weapons: Increasing male density promotes
#       sperm competition and reduces selection on arm strength in
#       a chorusing frog. Evolution 69: 2613-2624.
#       https://doi.org/10.1111/evo.12766

# Data originally downloaded on  6/6/2018 from
#   https://figshare.com/articles/Data_Paper_Data_Paper/3554424

# Data are included in the wildlifeR package and load from it
#     https://github.com/brouwern/wildlifeR
```

```
###############
## Packages
###############

library(wildlifeR)

###############
## Data set up
###############

# load frogarm data from Buzatto et al 2015
data("frogarms")

######################
## Data visualization
######################

# histogram of all data
hist(frogarms$sv.length)
```

```
######################
## Data analysis
######################

# unpaired t-test NOT using Welch's correction
## NOTE: assumes variation within each group EQUAL
t.test(sv.length ~ sex,   # model formula
       var.equal = TRUE,  # set variances to equal
         data = frogarms) # data

# unpaired t-test >>using<< Welch's correctin
t.test(sv.length ~ sex,
       var.equal = FALSE, # set variances to be unequal
         data = frogarms)
```

For more on organizing scripts see points 4 and 5 of "Eight things I do to make my open research more findable and understandable" at (DataColada](http://datacolada.org/69).

4.4 RMarkdown

[this section has not been completed]

- File
- New File
- R Markdown

A pop up menu will appear with lots of options; just click "ok."

Chapter 5

RMarkdown

[this chapter has not been written : ( ]

5.1 The “YAML” Header

5.2 Word processing

5.3 Code “chunks”

Part II

Getting Software & Data Into R

In this we'll discuss two of the most fundamental steps of data analysis: getting your hard-earned data into the @$*%# program you want to do you analysis in. This is often one of the most frustrating steps for beginners because R, like most data analysis tools that aren't spreadsheets, it seemingly very picky about how it wants to receive data. Luckily there's a method behind the madness, and also some handy tools in RStudio to help with this.

Most data starts off as a spreadsheet before it enters R. Loading spreadsheet data into R will be our end goal, but as a run up will step through several easier tasks that will highlight the core principles of getting data into R, in particular loading additional software into R (called packages) and loading the data in those packages into R. We'll also load data directly from the internet into R.

The chapters in this section are cover the following:

1. Loading R packages to get new software into R
2. Loading and looking at data from R packages
3. Loading packages from the software repository GitHub
4. Loading data from the internet
5. Loading spreadsheets as .csv files
6. Loading Excel spreadsheets

Chapter 6

Loading packages from CRAN

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter: lobrowR

6.1 Introduction

When you install R you get **base R**, which is the core set of functions, functionality, and some data sets. Base R however is surrounded by a universe of extensions built by statistician, programmers, academics and businesses that use R for analyses. A lot of R's functionality is found in these packages, including data sets, special plotting functions, and statistical tools for the analysis of complex data. Some of these are fairly standard and are downloaded along with base R and just need to be explicitly installed. Other have to be downloaded from the internet and installed. Most packages contain data in order to demonstrate what they do; working with data from packages will be covered in a later lesson.

This book relies heavily on an R package I've written called "wildlifeR" (https://brouwern.github.io/wildlifeR/) that contains the datasets used throughout the book, as well as some helpful R functions I've written.

Most R packages you'll use are stored on the CRAN website where you download R (https://cran.r-project.org/). R and RStudio have functions and tools for downloading and managing packages that we'll briefly introduce in this exercise.

Another place a package can be stored online is a code repository like GitHub. The wildlifeR package lives on GitHub and can be downloaded directly from there. Many packages on CRAN also occur on GitHub, especially if programmers are actively developing, updating, and managing the package. We'll cover downloading packages from GitHub in the next exercise.

6.1.1 Learning objectives

This exercise will introduce students to

1. the concept of an **R Package** (aka **library**)
2. how to load R packages using the library() function
3. the R plotting package **ggplot2**
4. cool add-ons to ggplot2, **ggpubr** and **cowplot**

6.1.2 Learning goals

By the end of this exercise students should be able to

- locate and download packages from the CRAN website using RStudio
- recognize the R functions used to download and install packages.

6.1.3 Functions & Arguements

- install.packages
- dependencies = TRUE
- library

6.1.4 Packages

- MASS
- ggplot2
- ggpubr
- cowplot

6.1.5 Potential hangups

- quoted vs. unquoted text (eg qplot vs. ggpubr syntax)

6.2 Loading packages that come with base R

What you download from CRAN is **base R** (also known as the **base distribution**). Many functions are called **base functions** because they are hard-wired into R.

6.3 OPTIONAL: What functions come with base R?

The following section is opitonal

If for some reason you want to see *all* the functions that come with base R, type this into the console and press enter. (ls stands for "list" and is a function we'll use more later).

```
ls("package:base")
```

As R has been developed there has also built up a cannon of tried and true packages that are downloaded automatically when you download R, but they aren't brought into R's working memory unless you tell R.

6.4 Optional: What packages come with base R?

> If you want to see all of the packages that come with base R, do this. library() is a function you will use a lot.

```
.libPaths("")
library()
```

One package that is part of this cannon is MASS, which stands for Modern Applied Statistics in S. “S” is the precursor to R, and MASS is the package that accompanies the book of the same name, which is one of the original books on S/R. (https://www.springer.com/us/book/9780387954578)

End optional section

6.4.1 Loading packages with the library() function

When a function is already downloaded to your computer, you explicitly load it into R’s working memory using the library() command.

```
library(MASS)
```

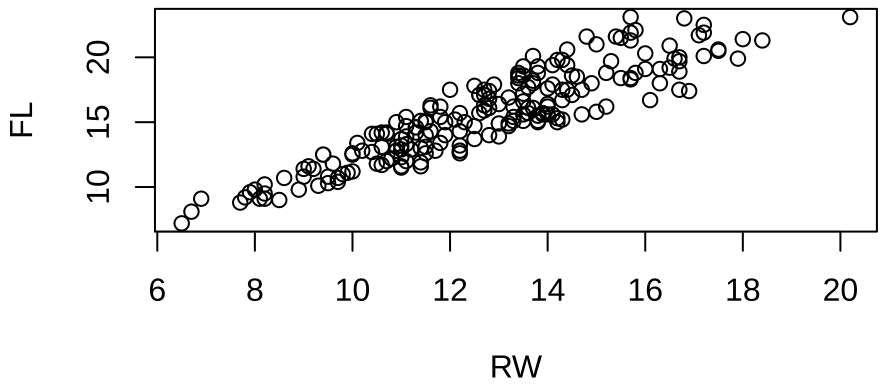
6.4.2 Preview: loading data from packages

Many packages have data. We can load data using the data() command.

```
data(crabs)
```

We plot data with the plot() command.

```
plot(FL ~ RW, data = crabs)
```



6.5 Load data from an external R package

Many packages have to be explicitly downloaded and installed in order to use their functions and datasets. Note that this is a **two-step process**: 1. Download package from internet 1. Explicitly tell R to load it

6.5.1 Step 1: Downloading packages with install.packages()

There are a number of ways to download packages. One of the easiest is to use the function install.packages(). Note that it might be better to call this "download.packages" since after you install it, you also have to load it!

Frequently in this book I will include install.packages(…) at the beginning of a lesson the first time we use a package to make sure the package is downloaded. Note, however, that if you already have downloaded the package, running install.packages(…) will download a new copy.

We'll download a package used for plotting called ggplot2, which stands for "Grammar of Graphics."

```
install.packages("ggplot2")
```

Often when you download a package you'll see a fair bit of red text, and sometime other things will pop up. Usually there's nothing of interest here, but sometimes you need to read things carefully over it for hints about why something didn't work.

6.6 Optional: Seeing all of your installed packages

The following section is optional

If for some reason you want to see everything you've downloaded, do this.

```
installed.packages()
```

End optional section

6.6.1 Step 2: Explicitly loading a package with library()

The install.packages() functions just saves the package software to R; now you need to tell R "I want to work with the package". This is done using the library() function. (Its called library because another name for packages is libraries)

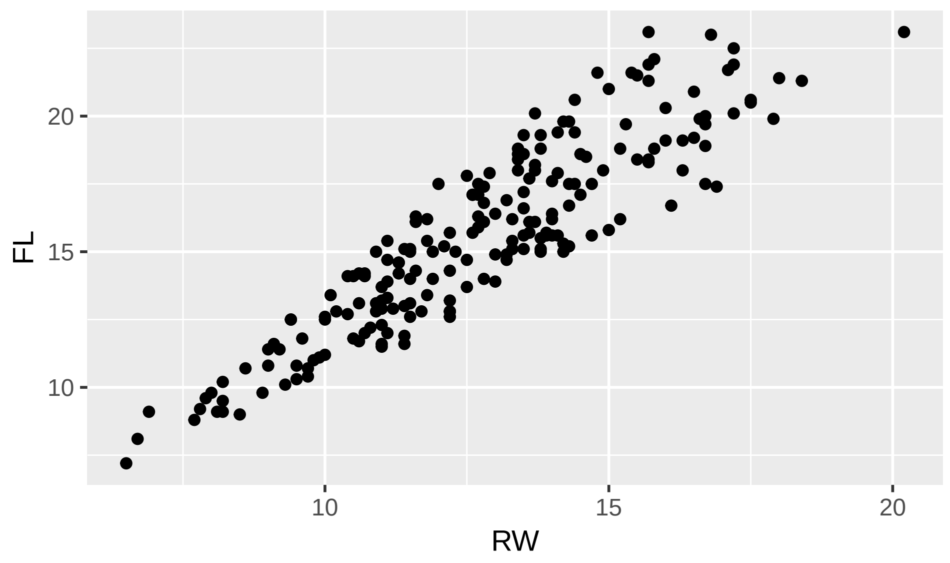
```
library(ggplot2)
```

6.7 OPTIONAL: Making a plot with ggplot

This section is optional

Now we can make a plot with the ggplot2 package we just downloaded, like using the qplot() function. (Note that the syntax is different than what we did above with plot()).*

```
qplot(y=FL, x= RW, data = crabs)
```



End opitional section

6.8 Downloading packages using RStudio

RStudio has a point-and-click interface to download packages. In the pane that says "Files, Plots, Packages, Help, Viewer" click on "Packages". When the panel shift below "Packages" it will say "Install, Update, Packrat." Click on "Install." (There might be a lag during this process as RStudio get info about your packages). In the pop up widow there will be a middle field "Packages" where you can type the name of your package. There's an auto-complete feature to help you in case you forget the name. Then click "install." Note that in the bottom right corner of the pop up is a checked box next to "Install dependencies." Leave that checked; more on that later.

6.9 Packages & their dependencies

R packages frequently use other R packages (which frequently use other R packages...). When an R package requires another package, its called a **dependency.** Depending on who and how the package was written up, dependencies might not be an issue or could be a problem.

As noted above when you download packages using RStudio's point and click interface there's a box that should be checked called "Install dependencies."

If you are using install.packages() there is an extra argument "dependencies = TRUE" that elicits the same behavior. I'll use this to get an add-on for ggplot2 called ggpubr.

```
install.packages("ggpubr",dependencies = TRUE)
```

We can then install this

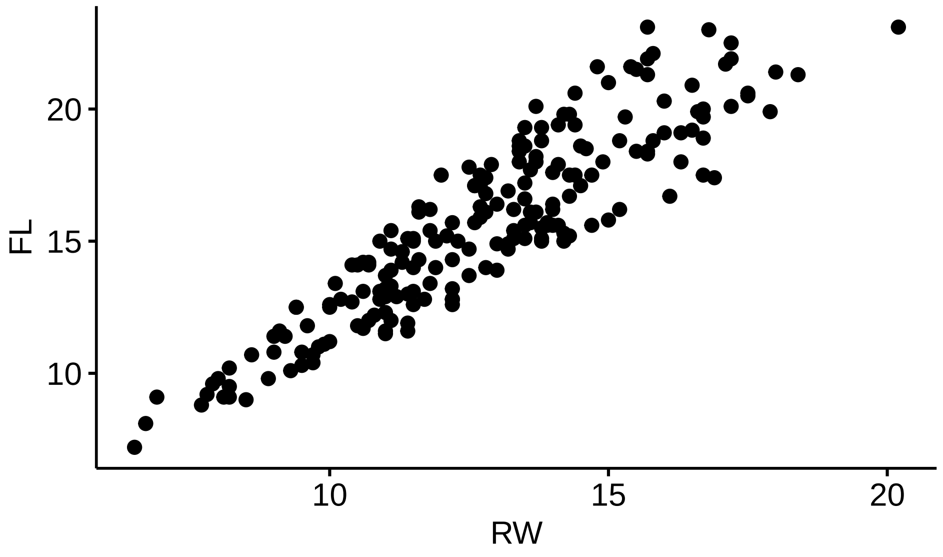
```
library(ggpubr)
```

6.10 Optional: Make a plot with ggpubr

This section is optional

ggpubr is an add on to ggplot. (This means that ggpubr has ggplot as a dependency). Note that the syntax for ggpubr function we use, ggscatter(), has a different syntax (again) than ggplot's qplot() function and base R's plot() function.

```
ggscatter(data = crabs,y = "FL", x = "RW") # use quotes!
```



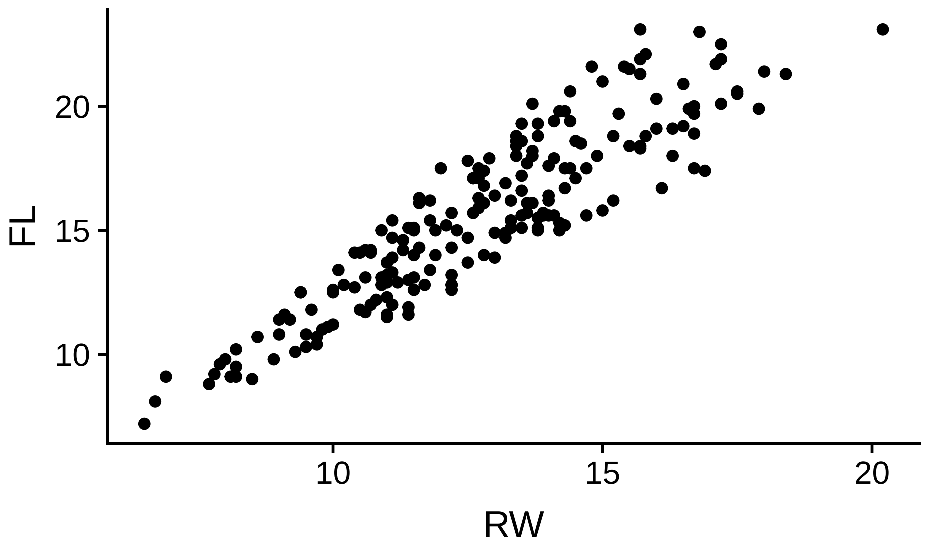
End optional section

6.11 Challenge

An another add-on to ggplot2 is cowplot, which stands for "Claus O. Wilke Plot". Download cowplot from CRAN using either the point-and-click method or **install.packages**, and then load it using **library**. Then run the following R code, which should make the plot below.

Note that "FL" and "RW" are NOT in quotation marks as they are for ggscatter()!

```
qplot(data = crabs, y = FL, x = RW) #no quotes!
```


FL
20
15
10
10
15
20
RW

Chapter 7

Loading data into R from a package

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter: lobrowR

7.1 Introduction

Working in R is all about working with data. There are many ways to get data into R, and RStudio has some helpful tools for this process. In this exercise we'll go over the common ways that data get's brought into R and how to download external packages to get datasets and functions. These include data

- pre-loaded in R
- loaded in R "packages"
- typed into a script
- loaded from a spreadsheet using RStudio's data import tools
- loaded from a spreadsheet using just R code

While discussing these various routes for data to get into R we'll also talk a bit about how R works with data and learn data related vocab.

7.1.1 Functions

- head(), tail()
- summary()

7.1.2 Datsets

- datasets::iris

7.1.3 Packages

7.1.4 Key terms

- package

- dataframe

7.2 Data pre-loaded in R

R comes with a number of datasets ready to use. A famous dataset frequently used in statistics is a set of measurements made on three species of irises and used to demonstrate some statistical principles by geneticist and statistician R.A. Fisher.

We can put the iris dataset into R's working memory using the data() command

```
data(iris)
```

We can see these data simply by type the word "iris" in the console and pressing enter. The dataset is too big for the screen probably and you'll just see a bunch of numbers flash by. You can get just a glimpse of the data by using the head() command, which will show you the first six or so rows of data.

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

(You can all use the tail() command to see the last 6 rows if you want.)

We can see that there are five rows of data. Three contain information about the length and width of the parts of the flower (Sepals and Petals) and the last holds the names of the species.

We can get a sense for these numbers by using the summary() command on the data, which will give us the mean and other summary statistics

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300  
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
##        Species  
##  setosa    :50  
##  versicolor:50  
##  virginica :50  
##                 
##                 
## 
```

Note that the last column doesn't contain numbers but rather names, so R counts up how many of each species name there is.

If we want to be reminded of the names of each column we can use the names() function

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

Looking at R data in the console isn't always very easy, so one thing you can do is use the View() command. This will bring up the data in a spreadsheet like viewer as a new tab in the script editor, similar to this.

```
pander::pander(iris[1:10,])
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa

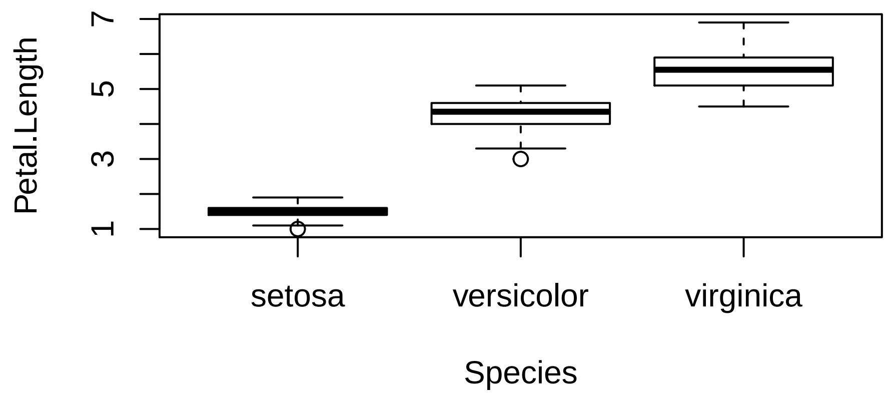
Note, however, that unlike a spreadsheet you cannot edit the data.

If you want to know more about a package, you can look at its help file, eg "?iris." These will often give you a fair bit of detail about what each column means, where the data are from, and may even have examples R functions applied to the data (though these can be rather obtuse, as is the case for the iris data).

7.2.1 Preview: Plotting boxplots

Plotting will be covered in depth in a subsequent exercise, but here's a glimpse of how we plot things in R:

```
plot(Petal.Length ~ Species, data = iris)
```



This code creates a series of **boxplots** of the petal lengths of each species of flower.

7.3 Loading data from R packages

Base R however is surrounded by a universe of extensions built by statistician, programmers, academics and businesses that use R for analyses. Some of these are fairly standard and are downloaded along with base R and just need to be explicitly installed. Other have to be downloaded from the internet and installed. Most packages contain data in order to demonstrate what they do.

7.3.1 Loading a package contained in base R

One package that is automatically downloaded but not automatically *installed* with base R is the "MASS" package, which stands for "Modern Applied Statistics in R"; S is the software that preceded R. We can install this package and make it functionally using the library() command

```r
library(MASS)
```

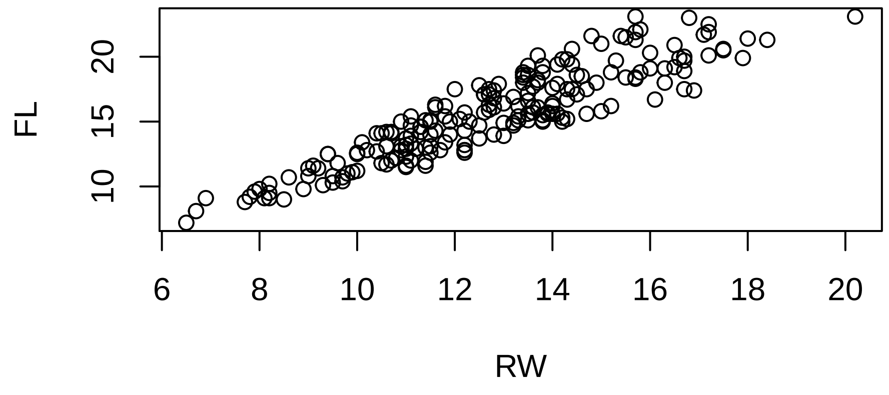
The MASS package has a biological dataset called "crabs" that you can put into working memory using data(crabs). We can then look at it using head(),View(), tail(), summary(), etc. We can find out more about the dataset using the help file, accessed via ?crabs

Question 1.What does the "FL" column mean in the crabs dataset? 1.What is the mean of the FL column?

7.3.2 Preview: Plotting scatter plot

We can plot the relationship between the FL and RW variables using a scatter plot.

```r
plot(FL ~ RW, data = crabs)
```



7.4 Learning about data in R

When data is being worked with in R, it lives in a place called the **workspace.** The workspace is not immediately transparent to you while working in R. It lives behind the scenes in what is essentially R's working memory. We can see what's on R's mind using the ls() command

```r
ls()
```

```
## [1] "crabs" "iris"  "x"
```

We can see our two datasets that we loaded using the data() command.

We can add new things to the work space using an R command like this

```r
my.mean <- mean(c(1,2,2))
```

Where "<-" is called the **assignment operator**. This function **assigns** the output of an R command or R function to an **R object** in R's working memory, the workspace.

We can check again what's on R's mind using a command ls(), which stands for "list"

```r
ls()
```

```
## [1] "crabs"   "iris"    "my.mean" "x"
```

We can see that we added my.mean. We can see what my.mean is by typing its name in to the console

```r
my.mean
```

```
## [1] 1.666667
```

We can also learn more about is using the is() command

```r
is(my.mean)
```

```
## [1] "numeric" "vector"
```

Here we get a big of R lingo: R tells use "numeric", which means it contain numeric data (numbers), and "vector", which is one of several types of R object

R objects can be just about anything. We can assign letter to an R object like this

```r
my.abc <- c("a","b","c")
```

Note that we have the letter each surrounded by quotes, and all 3 of them within c(…)

If you call up "my.abc" from the console, you will get back the three letter. Now see what is(my.abc) says

```r
is(my.abc)
```

```
## [1] "character"           "vector"              "data.frameRowLabels"
## [4] "SuperClassMethod"
```

There's a lot that comes out, but the first one says "character", indicating that yo have **character data** - data made up of text.

If you type ls() again what happens?

```r
ls()
```

```
## [1] "crabs"   "iris"    "my.abc"  "my.mean" "x"
```

We now see both of our R objects and the two datasets.

If we call is() on one of the dataset what do we is?

```r
is(crabs)
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

Several things get spit out, but the first one is important: "data.frame" Dataframes are fundamental units of analysis in R. Most of the data you will load into R and work within R will be in a dataframe.

Another function that tells about something in the the workspace is str(), which stands for structure. It provides info about what types of variables are in each column, and provides some sample output similar to head(), but oriented differently.

```
str(crabs)
```

```
## 'data.frame':    200 obs. of  8 variables:
##  $ sp   : Factor w/ 2 levels "B","O": 1 1 1 1 1 1 1 1 1 1 ...
##  $ sex  : Factor w/ 2 levels "F","M": 2 2 2 2 2 2 2 2 2 2 ...
##  $ index: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ FL   : num  8.1 8.8 9.2 9.6 9.8 10.8 11.1 11.6 11.8 11.8 ...
##  $ RW   : num  6.7 7.7 7.8 7.9 8 9 9.9 9.1 9.6 10.5 ...
##  $ CL   : num  16.1 18.1 19 20.1 20.3 23 23.8 24.5 24.2 25.2 ...
##  $ CW   : num  19 20.8 22.4 23.1 23 26.5 27.1 28.4 27.8 29.3 ...
##  $ BD   : num  7 7.4 7.7 8.2 8.2 9.8 9.8 10.4 9.7 10.3 ...
```

Note that the variables "sp", which stands for "Species", and "sex" are followed by the word "Factor." A **factor variable** is something that is or is summarized as discrete categories. For the species factor, there are two levels: the "B" species and the "O" species.

7.5 Load data from an external R package

Many packages have to be explicitly downloaded and installed in order to use their functions and datasets. Note that this is a **two step process**: 1. Download package from internet 1. Explicitly tell R to load it

7.5.1 Step 1: Downloading packages

There are a number of ways to install packages. One of the easiest is to use install.packages(). Note that it might be better to call this "download.packages" since after you install it, you also have to load it!

Well download a package used for plotting called ggplot2, which stands for "Grammar of graphics"

```
install.packages("ggplot2")
```

Often when you download a package you'll see a fair bit of red text. Usually there's nothing of interest hear, but sometimes you need to read over it for hints about why something didn't work.

7.5.2 Step 2: Explicitly loading a package

The install.packages() functions just saves the package software to R; now you need to tell R "I want to work with the package". This is done using the library() function. (Its called library because another name for packages is libraries)

```
library(ggplot2)
```

ggplot2 has a dataset called "msleep" which has information on the relationship between the typical size of a species and its brain weight, among other things

We load the data actively into R's memory using data(), and can look at the column names using names()

```
data(msleep)
names(msleep)
```

```
##  [1] "name"         "genus"        "vore"         "order"
##  [5] "conservation" "sleep_total"  "sleep_rem"    "sleep_cycle"
##  [9] "awake"        "brainwt"      "bodywt"
```

We can now explore this data set as before using summary(), str(), etc.

Another useful command when you are working with a new dataset is dim(). This tells you the dimension of the dataframe

```
dim(msleep)
```

```
## [1] 83 11
```

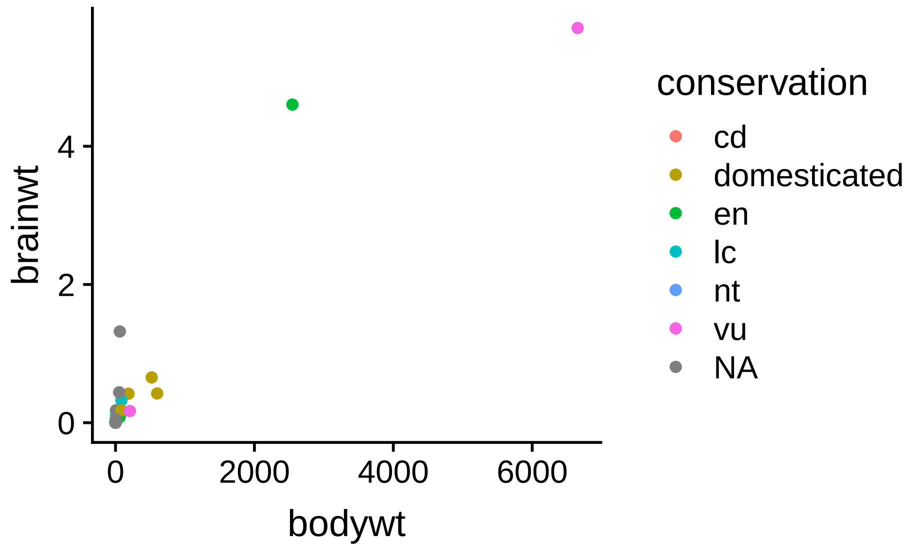
7.5.3 Preview: plotting with ggplot2

ggplot2 is a powerful plotting tool that has become standard among scientists, data scientists, and even journalists. Here's a quick way to make a plot in ggplot2 using its qplot() function (qplot = quick plot, not to be confused with qqplot). Note that the qplot() function only works if you have ggplot2 downloaded and installed.

A powerful aspect of ggplot is the fact that it can easily be used to modify plots. Here, we use the **arguement** "color =" to color code the data points based on their IUCN red list status.

```
qplot(y = brainwt, x = bodywt, data = msleep, color = conservation)
```

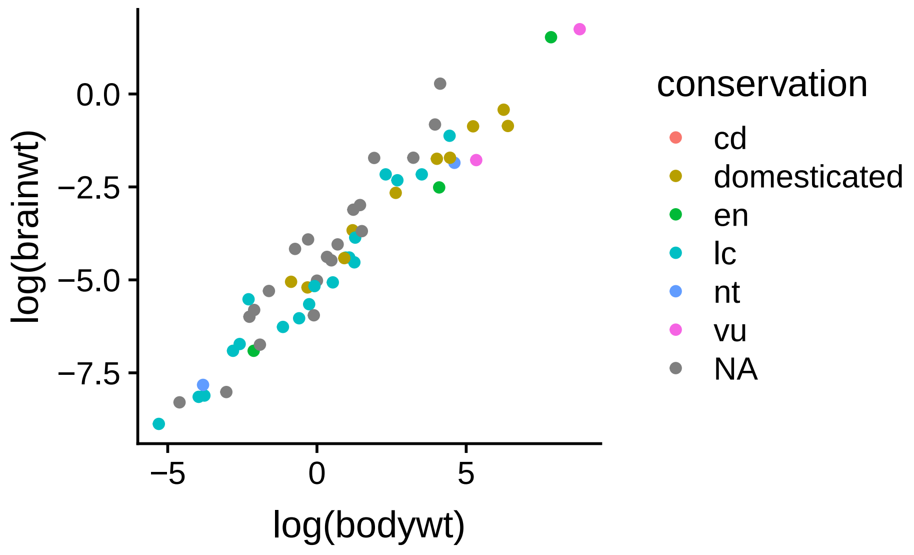
```
## Warning: Removed 27 rows containing missing values (geom_point).
```



The animals in this data vary in size from mice to elephants and so a lot of the data points are scrunched together. A trick to make this easier to see is to take the log of the brainwt and bodywt variable. In R, we can do this on the fly like this using the log() command

```
qplot(y = log(brainwt), x = log(bodywt), data = msleep, color = conservation)
```

```
## Warning: Removed 27 rows containing missing values (geom_point).
```



7.6 Loading data from an R script

So far we have only looked at dataset that are already formatted into **dataframe** by somebody for us. Now we want to look at how to set up datasets ourselves. When datasets are small its possible to enter them more or less directly into R by typing out all of the numbers in a script. This only works well for when datasets are small; even when datasets are small its best to keep them separate from your R code in a spreadsheet file. However, its useful to know how to load data this way; even when an exercise in this book loads data from a package or spreadsheet I will also often include the code to load it directly just in case there is an issue with download the package or file.

7.6.1 The eagles have landed - in your R workspace

In a subsequent exercise we will practice using data on the number of eagles in Pennsylvania and other states in the USA. We can load this data into R by making R objects, and then turning these objects into a dataframe.

7.6.1.1 Step one: Build R objects

First, we'll use the assignment operator ("<-") to create an R object called "year" that lists the years from 1980 through 2015 for which the number breeding pairs of eagles in Pennsylvania, USA, is known.

```
year <- c(1980,1981,1982,1983,1984,1985,1986,1987,1988,1989,
          1990,1991,1992,1993,1994,1995,1996,1997,1998,1999,
          2000,2001,2002,2003,2004,2005,2006,2007,2008,2009,
          2010,2011,2012,2013,2014,2015,2016)
```

A quick trick to do this much fast is

```
year <- c(1980:2016)
```

Second, we'll create an object called "eagles" with the number of breeding pairs (male and females paired up for making baby eagles) recorded each year. Note that most years in the 1980s are skipped because there is not data available. When data are missing we use NA. (Note that this is just NA, with not quotes around it).

```
eagles <-  c(3, NA, NA, NA, NA, NA, NA,NA,NA,NA,
             7,  9, 15, 17, 19, 20, 20,23,29,43,
             51,55, 64, 69, NA, 96,100,NA,NA,NA,
             NA,NA, NA, NA,252,277, NA)
```

7.6.1.2 Step two: Build dataframe

We can then turn these two separate R objects into a dataframe

```
eagle.df <-data.frame(year, eagles)
```

7.6.2 Looking at the eagle data

We can check that we have this R object by using the **ls()** command.

```
ls()
```

```
## [1] "crabs"    "eagle.df" "eagles"   "iris"     "msleep"   "my.abc"
## [7] "my.mean"  "x"        "year"
```

And we can confirm that its a dataframe using **is()**

```
is(eagle.df)
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

summary() will give us basic info on PA's eagles

```
summary(eagle.df)
```

```
##       year          eagles
##  Min.   :1980   Min.   :  3.00
##  1st Qu.:1989   1st Qu.: 18.00
##  Median :1998   Median : 29.00
##  Mean   :1998   Mean   : 61.53
##  3rd Qu.:2007   3rd Qu.: 66.50
##  Max.   :2016   Max.   :277.00
##                 NA's   :18
```

Note that in the "eagles" columns it tells you the number of NAs (missing values). The summary() readout quickly tells us that the eagle population has changed dramatically.

7.6.2.1 Preview: Plotting the Eagle D

We can plot the data in ggplot2 using qplot(). However, there is an excellent package that adds additional functionality to ggplot called ggpubr. This is fairly common in R: you have packages that add functions to R, and packages that add functions to other packages.

We can install ggpubr using install.packages(). Note that the name of the package, ggpubr, is in quotes.

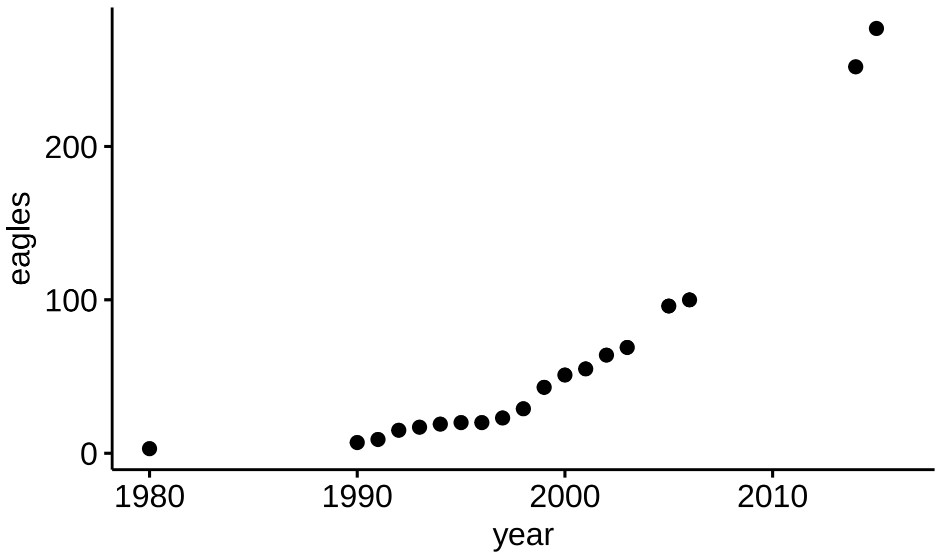
```
install.packages("ggpubr")
```

ggpubr requires *another package*, magrittr, which R tells you about in red text. When a package requires another package, its called a **dependency** because one package relies on another. ggpubr has magrittr as a dependency; ggpubr modifies ggplot2, so ggpubr has ggplot2 as a dependency.

Occasionally you might try to load a package and it won't automatically install or download the dependency, usually because its not yet downloaded. If this happens with magrittr we would just have to download it using "install.packages("magrittr")".

Once we have ggpubr loaded we can plot the eagle data using the handy function **ggscatter()**

```
ggscatter(dat = eagle.df, y = "eagles", x= "year")
```

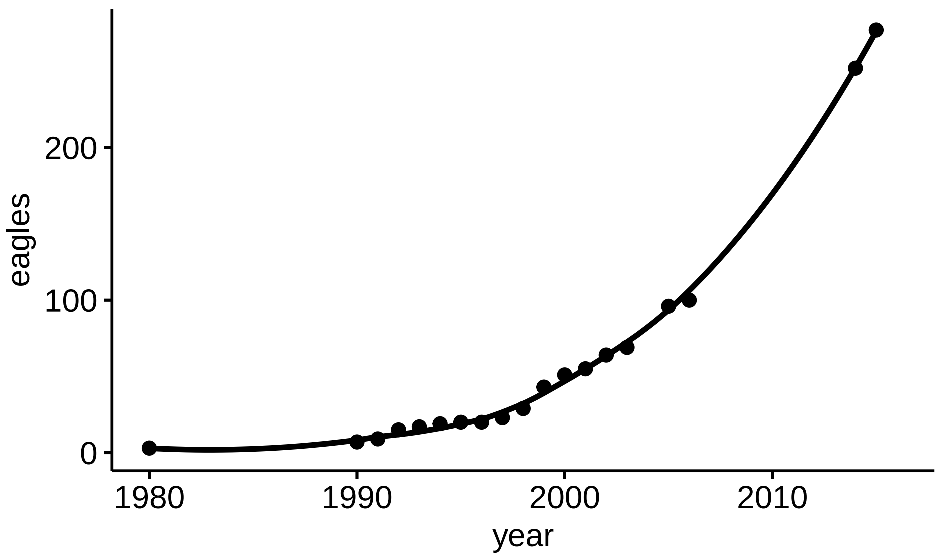


7.7 Challenge

We can add a **smoother** to the plot by addin add = "loess".

```
## Warning: Removed 18 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 18 rows containing missing values (geom_point).
```


eagles
200
100
0
1980
1990
2000
2010
year

Chapter 8

Loading packages & data from GitHub

Nathan Brouwer, Phd
brouwern@gmail.com
https://github.com/brouwern
?

8.1 Introduction

GitHub is an online platform for hosting and sharing code. More formally it is called a **software repository**. It is very popular with software developers, especially those creating open-source applications, and has also been adopted whole-hearted by many data scientists and data analysts.

GitHub has many features and uses. One of the most basic ones is to use GitHub like Dropbox to R backup copies of code on GitHub. GitHub also can act like a kind of web server to host websites, online books like this one, and provide access to open source software. Many people working on R packages use GitHub to host their package while its being developed or expanded. When a package is finished, it often is then submitted to CRAN, and the version on GitHub is used as the **developement version** where new features are being developed and tested.

You can access packages on GitHub to get the newest version before something has been submitted to CRAN, or packages that haven't or maybe will never end up on CRAN. This book relies on a package I've written called **wildlifeR** for datasets and some functions. In this short exercise we'll download a package from CRAN we need to interact with GitHub, and then download wildlifeR. We'll also go to the wildlifeR website to learn more about the package.

8.1.1 Learning objectives

8.1.2 Learning goals

8.1.3 Functions & Arguements

- library
- devtools::install_github
- scatter.smooth

- $

8.1.4 Packages

- devtools
- wildlifeR

8.1.5 Potential hangups

- We'll use the "$" operator to tell scatter.smooth() what to plot, which is different than how ggpubr and ggplot2 work; sigh…

8.2 Accessing GitHub using devtools

The devtools package is used by many people who write R packages and includes a function for downloading from GitHub

```r
install.packages("devtools", dependencies = TRUE) # [ ]
```

devtools has a lot of dependencies so this might take a while.

Once everything is downloaded, load the package explicitly with library()

```r
library(devtools)
```

8.3 Downloading the wildlifeR package with install_github()

My github site is at https://github.com/brouwern and the code for wildlifeR is https://github.com/brouwern/wildlifeR. You can access the files directly if you want, but that isn't necessary. We can download the package just like it was on CRAN using install_github(). You'll probably see some red text and a LOT of black text as install_github() talks with GitHub.

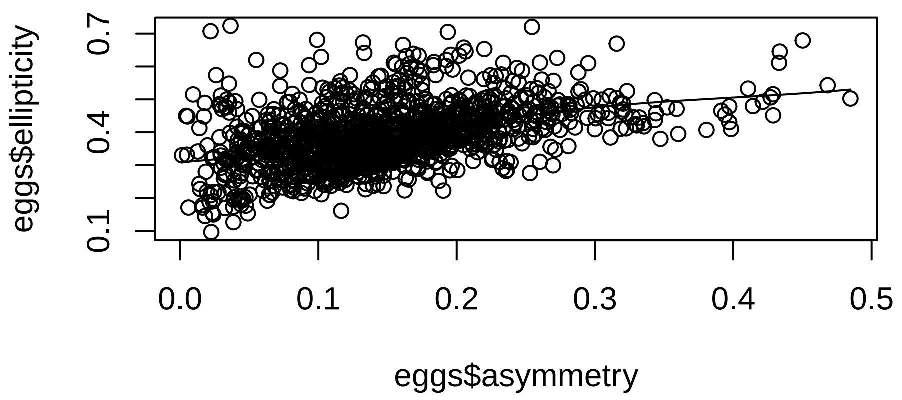
```r
install_github("brouwern/wildlifeR")
```

Now we can put it all explicitly into memory

```r
library(wildlifeR) # [ ]
```

OPTIONAL: Accessing data from wildlifeR One of the datasets in wildlifeR is called "eggs." It has data from a paper by Stoddard et al. (2017) in Science called [Avian egg shape: Form, function, and evolution.] (http://science.sciencemag.org/content/356/6344/1249). We can plot the relationship between egg asymmetry and ellipticity using the base R function scatter.smooth(), which draws a type of regression line through the data for us (Note that the syntax for scatter.smooth() is, sadly, different than plot() and other plotting functions…).

```r
scatter.smooth(eggs$asymmetry, eggs$ellipticity)
```



8.4 The wildlifeR packge webiste

Some packages have websites that summarize the package contents. If you visit https://brouwern.github.io/wildlifeR/ you can find out information on each dataset and function under the "Reference" tab, and see how the datasets and functions are used under the "Articles" tab.

Chapter 9

Loading data from the internet

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter: lobrowR

9.1 Introduction

Its possible to download data directly from the internet, including

- Spreadsheets directly posted online as .csv or .txt
- Spreadsheets contained within a GitHub repository, including a package
- Google Sheets
- and many other formats

In this short exercise we'll download some data that is stored as a raw .csv file within the inner workings of the widlifeR package.

9.1.1 Learning Goals & Outcomes

By the end of this lesson students should be able to download basic data sources from the internet using getURL() from the RCUrl package.

9.1.2 Functions & Arguements

- RCurl::getURL()
- scatter.smooth

9.1.3 Packages

- RCurl install.packages()

9.1.4 Potential hangups

- Bad internet connection
- Firewall problems

9.2 Downloading a .csv file using getURL()

The package RCurl provides functions for accessing online material. First we need the package

```
install.packages("RCurl", dependencies = TRUE) # [ ]
```

As always, once we install a package we need to **really** install it with library(). (You might see some red text as RCurl loads up some of its **dependencies**)

```
library(RCurl) # [ ]
```

We then use the getURL() to prep the info we need for downloading that .csv we want.

The file we want is "eaglesWV.csv". It is located at this rather long URL: https://raw.githubusercontent.com/brouwern/wildlifeR/master/inst/extdata/eaglesWV.csv

First, we'll use the "<-" assignment operator to store the shortened URL in an R object. Be sure to put the URL in quotes.

```
eaglesWV.url <- "https://raw.githubusercontent.com/brouwern/wildlifeR/master/inst/extdata/eaglesWV.csv"
```

Next we'll use the getURL() function to set things up, storing the info in a new object "eaglesWV.url_2" (note the "_2" on the end).

First, get the URL from the URL-containing object we just made.

```
eaglesWV.url_2 <- getURL(eaglesWV.url)
```

Now use read.csv() to actually get it.

```
eaglesWV_2 <- read.csv(text = eaglesWV.url_2)
```

We can preview the downloaded dataset using summary() or any other command we want

```
summary(eaglesWV_2)
```

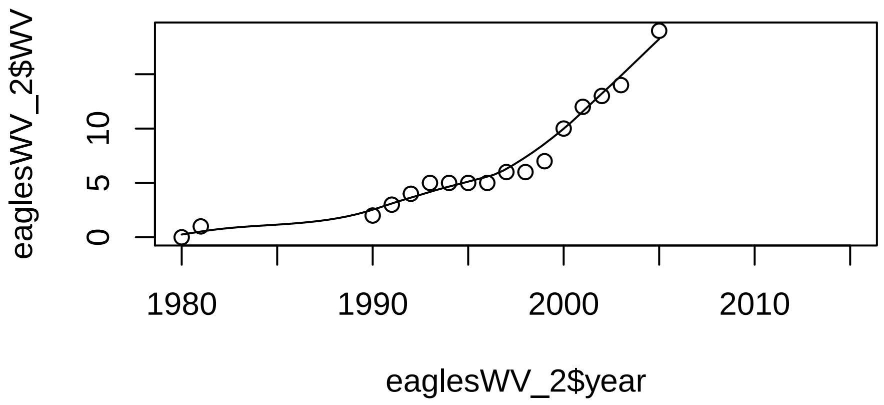
```
##       year            WV
##  Min.   :1980   Min.   : 0.000
##  1st Qu.:1994   1st Qu.: 4.000
##  Median :2001   Median : 5.000
##  Mean   :2001   Mean   : 6.882
##  3rd Qu.:2008   3rd Qu.:10.000
##  Max.   :2015   Max.   :19.000
##                 NA's   :12
```

9.3 OPTIONAL: Plotting West Virginia Eagle Data

This section is optional

Thankfully, eagles having been increasing exponentially in West Virginia since the 1980s.

```
scatter.smooth(y = eaglesWV_2$WV,x = eaglesWV_2$year)
```



End optional section

Chapter 10

Loading data from .csv files into RStudio

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter: lobrowR

10.1 Introduction

We will be working with data from Table 2 of Medley and Clements (1998). (This data is featured in the ecological stats book by Quinn and Keough (2002), though I'm not a fan of how they analyse it.) The paper looks at how diatoms(photosynthesizing microorganisms known for their silica shells) are impacted by water quality in mountain streams.

10.1.1 Learning goals

10.1.2 Learning objectives

By the end of this lesson students will be able to

- Download raw data files by hand from the internet
- Load .csv files using the R command read.csv()
- Load .csv files using RStudio's point-and-click interface

10.1.3 R packageas

10.1.4 R commands

- read.csv
- View
- setwd
- getwd
- list.files
- read.csv

- ls
- dim
- names
- summary

10.1.5 Files

- Medley1998.csv

10.1.6 Potential Hangups

10.1.7 References

Medley & Clements. 1998. Responses of diatom communities to heavy metals in streams: The influence of longitudinal variation. [Ecological Applications 8:631-644.] (https://www.jstor.org/stable/2641255)

Quinn & Keough. 2002. Experimental design and data analysis for biologists.. A pdf version of the book is available online for free..

10.2 Preliminary step: download a .csv file

To load a .csv file into R we first need a .csv file to load. The data we'll be working with can be downloaded from GitHub. First, go to the following link (it happens to be an obscure subfolder of the wildlifeR package)

https://github.com/brouwern/wildlifeR/tree/master/inst/extdata

Next, locate the file Medley1998.csv

Click on it; a table will show up.

This table is formatted to look nice on a webpage (using some HTML that GitHub impose on the file). We want the raw file itself. To get it we need to click on the **"Raw"** tab.

We will then see what looks like a text document against a white background with no formatting of any kind. We can now download the file by following these steps.

Either

- Use the shortcut **Control + S** to "Save as" the file

Or

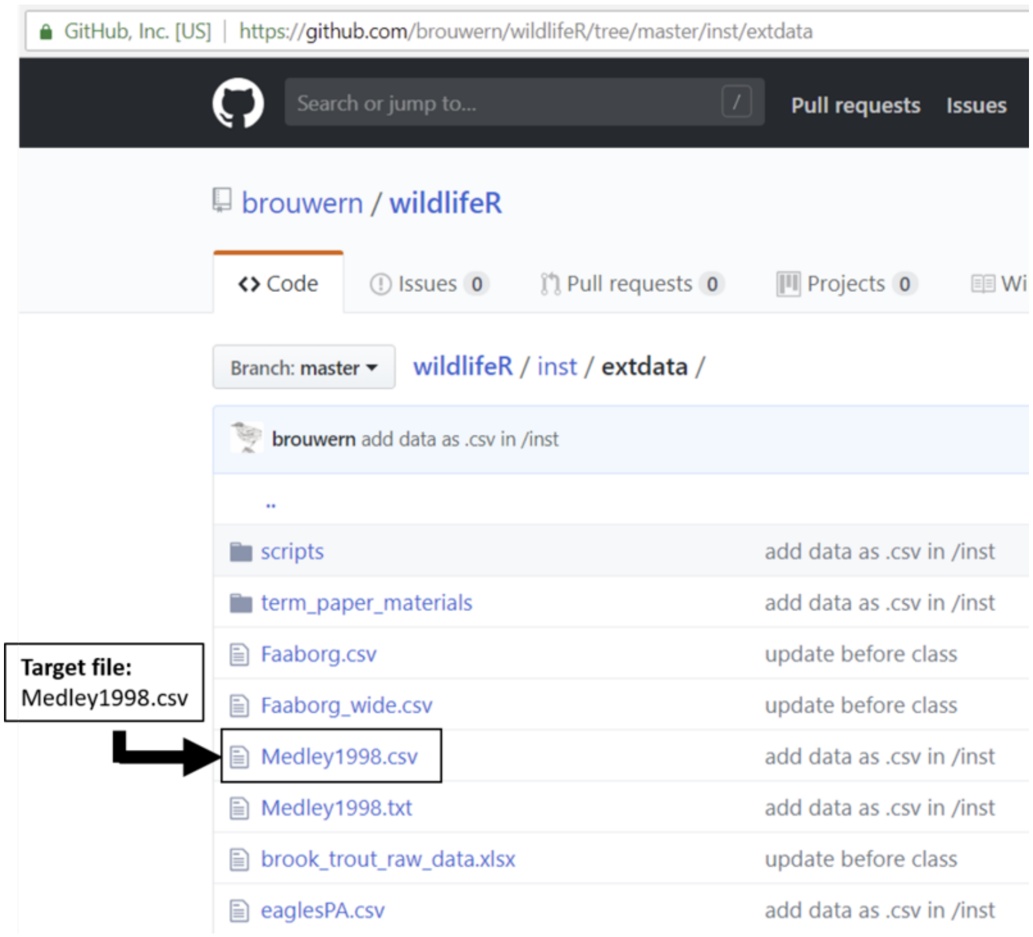
- Right click (on Mac:…)
- "Save link as" (or the equivalent)

Then save the file to a location you know you can find, such as

- Documents
- Desktop
- Your network profile drive

Note that if you try to "Save as…" anything else but the white-screen **raw text file** you will run into problems.

After you download the file, open up Excel or another spreadsheet program and open up the file to confirm that what you downloaded is just a set of numbers. I you see long lines of text you might have accidentally downloaded the HTML-formatted version of the file. Make sure you are downloading the very plain version of the file from the totally blank white screen.


GitHub, Inc. [US] | https://github.com/brouwern/wildlifeR/tree/master/inst/extdata

brouwern / wildlifeR

Branch: master ▾ wildlifeR / inst / extdata /

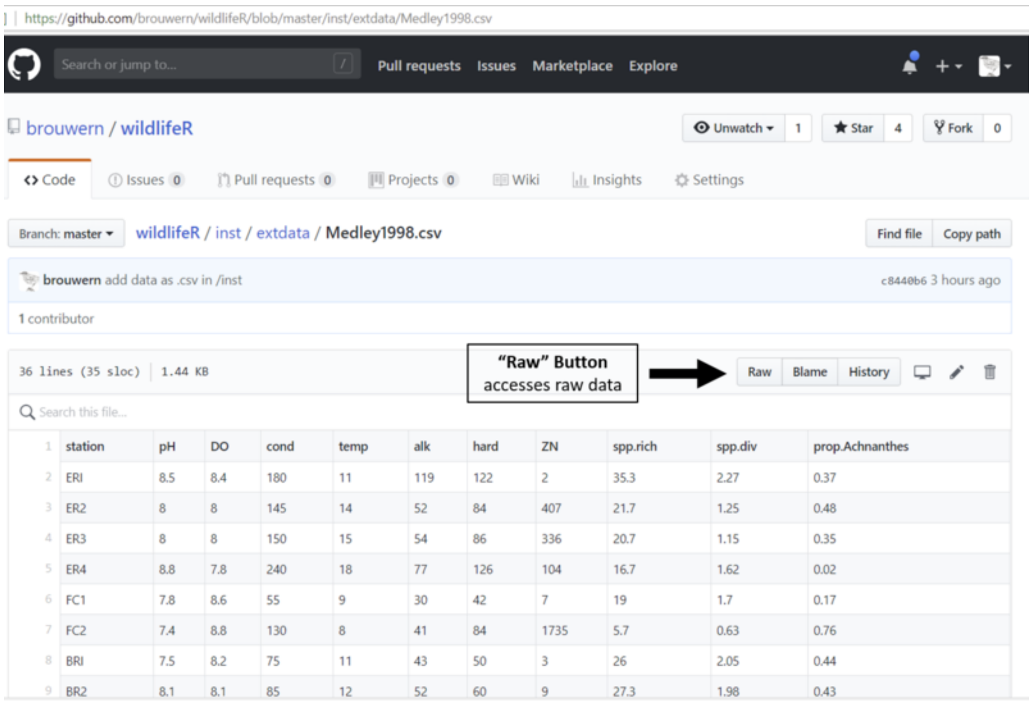
brouwern add data as .csv in /inst

..

scripts	add data as .csv in /inst
term_paper_materials	add data as .csv in /inst
Faaborg.csv	update before class
Faaborg_wide.csv	update before class
Medley1998.csv	add data as .csv in /inst
Medley1998.txt	add data as .csv in /inst
brook_trout_raw_data.xlsx	update before class
eaglesPA.csv	add data as .csv in /inst

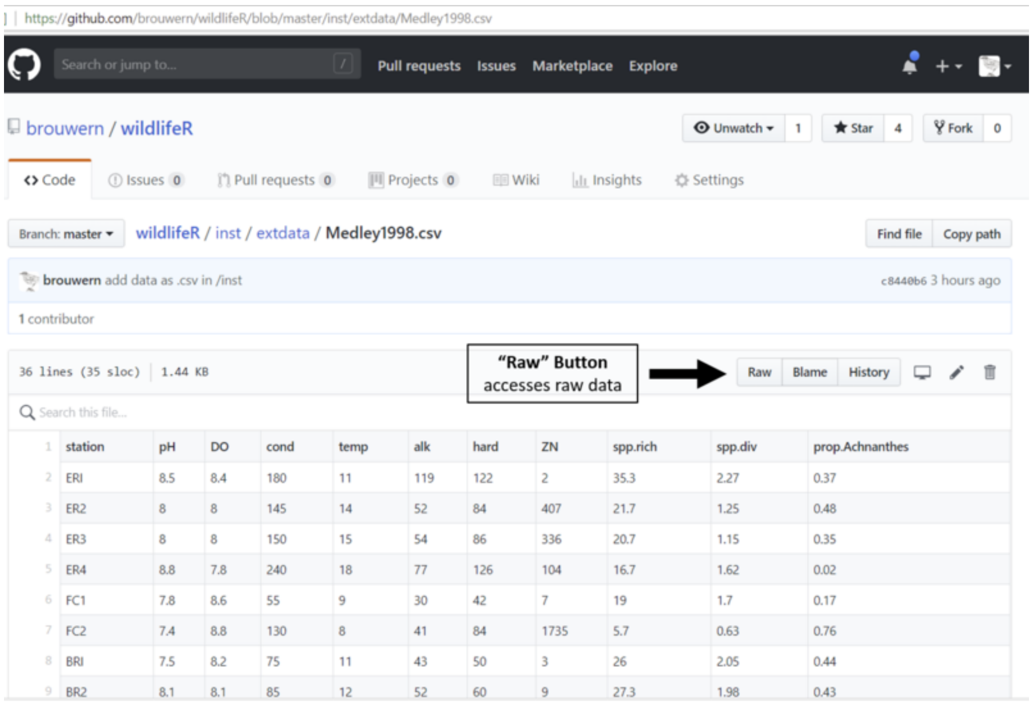
Target file:
Medley1998.csv

Figure 10.1: A list of files stored on GitHub.



station	pH	DO	cond	temp	alk	hard	ZN	spp.rich	spp.div	prop.Achnanthes
ERI	8.5	8.4	180	11	119	122	2	35.3	2.27	0.37
ER2	8	8	145	14	52	84	407	21.7	1.25	0.48
ER3	8	8	150	15	54	86	336	20.7	1.15	0.35
ER4	8.8	7.8	240	18	77	126	104	16.7	1.62	0.02
FC1	7.8	8.6	55	9	30	42	7	19	1.7	0.17
FC2	7.4	8.8	130	8	41	84	1735	5.7	0.63	0.76
BRI	7.5	8.2	75	11	43	50	3	26	2.05	0.44
BR2	8.1	8.1	85	12	52	60	9	27.3	1.98	0.43

Figure 10.2: An HTML .csv datafile store on GitHub. The raw file can be accessed by clicking on the Raw tab.



station	pH	DO	cond	temp	alk	hard	ZN	spp.rich	spp.div	prop.Achnanthes
ERI	8.5	8.4	180	11	119	122	2	35.3	2.27	0.37
ER2	8	8	145	14	52	84	407	21.7	1.25	0.48
ER3	8	8	150	15	54	86	336	20.7	1.15	0.35
ER4	8.8	7.8	240	18	77	126	104	16.7	1.62	0.02
FC1	7.8	8.6	55	9	30	42	7	19	1.7	0.17
FC2	7.4	8.8	130	8	41	84	1735	5.7	0.63	0.76
BRI	7.5	8.2	75	11	43	50	3	26	2.05	0.44
BR2	8.1	8.1	85	12	52	60	9	27.3	1.98	0.43

Figure 10.3: A raw .csv datafile stored on GitHub. It can be downloaded By using Crtl+S or right clicking and selecting Save As

10.3 Set the "working directory" ("WD") in RStudio

We will now take the data we saved as a .csv file and load it into R. This can be tricky. First we need to tell R exactly where the file is by setting the **working directory**.

Follow these steps:

- Click on "Session" on the main menu
  - on the menu: "File, Edit, Code, View, Plots, **Session**, …"
- Click on "Set working directory"
- Select "Choose Directory"
- Select your computer's Documents folder or wherever else you chose to save the file.
- Select the directory & click "Open"
- Note that the command "setwd()" shows up in the console followed by the location of the directory you selected

You can set your working directory to be anywhere on the computer. It is essential to make sure that the csv file you want to load into R is in your working directory.

Depending on the location you chose you might just see "~/" or some other shorthand.

10.4 Check the working directory with getwd()

You can confirm where you are at using the command **getwd()**; this can be handy if you're not sure that you did things correctly or if R didn't output what you expected.

```
getwd() # [ ]
```

```
## [1] "C:/Users/lisanjie/Documents/1_R/git/git-teaching/teaching_2018_2019/2018_fall/biostats_fall_2018
```

Here, even though when set the working directory R originally just displayed "setwd("~/")", I can now confirm that I'm in my documents folder.

10.5 Check for the file you downloaded with list.files()

You can see what's in your working directory using the command **list.files()**. Depending on how many files you have this could be a very long list. I have 40-ish files and so won't display them.

```
list.files() # [ ]
```

If you have a ton of files being printed out you can narrow things down by telling R a text pattern to screen for.

```
list.files(pattern = "csv") # [ ]
```

```
## character(0)
```

If the file wasn't successful downloaded R will just give you a cryptic message like this.

```
list.files(pattern = "xxxx")
```

This means the file isn't' there and you need to redo the download to make sure either i)the file actually downloaded and ii)file is saved where you want it to be.

What we want to see is this

```
## character(0)
```

10.6 OPTIONAL Interacting with R via the console or the source viewer

This section is optional

You can enter R commands directly into the console, or type them into a **script file** in the **source viewer** and then execute.

If you've just been using the console try this:

- Click on the source viewer pane in RStudio
- Type "getwd()" in the source viewer
- Click on the "Run" button in the upper Right part of the pane
- The getwd() command is sent over to the console and executed

End optional section

10.7 Loading data into R using read.csv()

Copy and paste the .csv file name from the console into the source viewer then Execute the command "read.csv(file ="Medley1998.csv")". You can type it but you must be careful to have NO TYPOS. R is unforgiving when it comes to typos.

If you've done it correctly you'll see the data table printed out in the console (I show only some of the output).

```
read.csv(file = "Medley1998.csv")
```

```
##   station  pH  DO cond temp alk hard   ZN spp.rich spp.div prop.Achnanthes
## 1     ERI 8.5 8.4  180   11 119  122    2     35.3    2.27            0.37
## 2     ER2 8.0 8.0  145   14  52   84  407     21.7    1.25            0.48
## 3     ER3 8.0 8.0  150   15  54   86  336     20.7    1.15            0.35
## 4     ER4 8.8 7.8  240   18  77  126  104     16.7    1.62            0.02
## 5     FC1 7.8 8.6   55    9  30   42    7     19.0    1.70            0.17
## 6     FC2 7.4 8.8  130    8  41   84 1735      5.7    0.63            0.76
```

You must have the file name in quotation marks and include the ".csv". *Any* small error will cause things to not work.

Here are examples of mistakes that *won't work* (no matter how much you cuss at it.)

```
read.csv(file = Medley1998.csv)      #missing quotes " "
read.csv(file = "Medley1998.csv")    #missing .csv
read.csv(file "Medley1998.csv")      #missing =
```

Note that R returns error messages in red, but they aren't necessarily very helpful in figuring out what the problem actually is. This is an unfortunate feature of R, and reading error messages is a skill that must be learned.

10.7.1 Load data into an R "object"

Now type this: "med98 <- read.csv(file ="Medley1998.csv")". The "<-" is the **assignment operator**. What happens when you execute this command?

```
med98 <- read.csv(file = "Medley1998.csv") [ ]
```

It might actually look like not much has happened. But that's good! It means the data has successful been loaded into R. You have "assigned" the data from your file to the "object" named "med98"

10.7.2 The assignment operator "<-"

"<-" is called the "assignment operator". It is a special type of R command.

"<" usually shares The comma key. Type "shift + ," To get it.

If you type just "med98" and execute it as a command, what happens?

```
med98
```

```
##   station  pH  DO cond temp alk hard   ZN spp.rich spp.div prop.Achnanthes
## 1     ERI 8.5 8.4  180   11 119  122    2     35.3    2.27            0.37
## 2     ER2 8.0 8.0  145   14  52   84  407     21.7    1.25            0.48
## 3     ER3 8.0 8.0  150   15  54   86  336     20.7    1.15            0.35
## 4     ER4 8.8 7.8  240   18  77  126  104     16.7    1.62            0.02
## 5     FC1 7.8 8.6   55    9  30   42    7     19.0    1.70            0.17
## 6     FC2 7.4 8.8  130    8  41   84 1735      5.7    0.63            0.76
```

You should see the entire dataset spit out in the console (I've just shown the top part).

Now execute the list command **ls()**. You should now see "med98" shown in the console.

```
ls()
```

```
##  [1] "crabs"           "eagle.df"        "eagles"          "eaglesWV.url"
##  [5] "eaglesWV.url_2" "eaglesWV_2"      "iris"            "med98"
##  [9] "msleep"          "my.abc"          "my.mean"         "x"
## [13] "year"
```

This means that the **object** you assigned your data is now in your **"workspace."** The workspace is what I call the working memory of R.

We can learn about the med98 data using command like dim(), names() and summary().

How big is the dataset overall?

```
dim(med98)
```

```
## [1] 34 11
```

How man columns are there?

```
names(med98)
```

```
##  [1] "station"         "pH"              "DO"
##  [4] "cond"            "temp"            "alk"
##  [7] "hard"            "ZN"              "spp.rich"
## [10] "spp.div"         "prop.Achnanthes"
```

Are any of the variables categorical?

```r
summary(med98)
```

```
##     station          pH              DO              cond       
##  AR2    : 1   Min.   :6.700   Min.   :6.800   Min.   : 40.00  
##  AR3    : 1   1st Qu.:7.425   1st Qu.:7.500   1st Qu.: 76.25  
##  AR5    : 1   Median :7.900   Median :7.600   Median :100.00  
##  AR8    : 1   Mean   :7.841   Mean   :7.794   Mean   :116.76  
##  ARI    : 1   3rd Qu.:8.200   3rd Qu.:8.175   3rd Qu.:150.00  
##  BR2    : 1   Max.   :8.800   Max.   :8.800   Max.   :240.00  
##  (Other):28                                                   
##       temp            alk              hard              ZN        
##  Min.   : 8.00   Min.   : 10.00   Min.   : 10.00   Min.   :   2.0  
##  1st Qu.:11.00   1st Qu.: 28.50   1st Qu.: 45.00   1st Qu.:  24.0  
##  Median :12.50   Median : 46.50   Median : 62.00   Median :  54.0  
##  Mean   :13.06   Mean   : 46.38   Mean   : 66.76   Mean   : 177.3  
##  3rd Qu.:15.00   3rd Qu.: 64.00   3rd Qu.: 90.50   3rd Qu.: 213.2  
##  Max.   :21.00   Max.   :119.00   Max.   :126.00   Max.   :1735.0  
##                                                                    
##     spp.rich        spp.div      prop.Achnanthes 
##  Min.   : 5.70   Min.   :0.630   Min.   :0.0200  
##  1st Qu.:18.77   1st Qu.:1.377   1st Qu.:0.2125  
##  Median :22.85   Median :1.855   Median :0.3900  
##  Mean   :22.42   Mean   :1.694   Mean   :0.3756  
##  3rd Qu.:26.82   3rd Qu.:2.058   3rd Qu.:0.4950  
##  Max.   :42.00   Max.   :2.830   Max.   :0.7600  
## 
```

10.8 Optional: Plot the Mendley 1998 data

The following section is optional

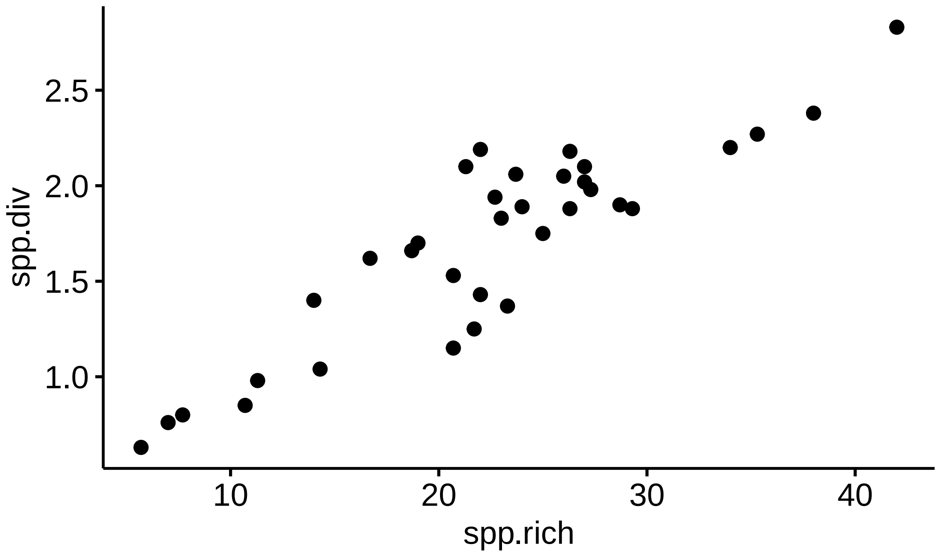
As we'll discuss in depth in a later section on plotting , one reason why the **ggplot** and **ggpubr** packages are so powerful is because they can easily plot things in good color schemes. We can make a basic scatter plot like this to show the positive correlation between Diatom species richness (the raw number of species identified in a given sample) on the x axis and species diversity on the y axis.

First, load the ggpubr package using the library() command. Note that you might get some output in red text telling you about the packages; it looks scary but its not.

```r
library(ggpubr)
```

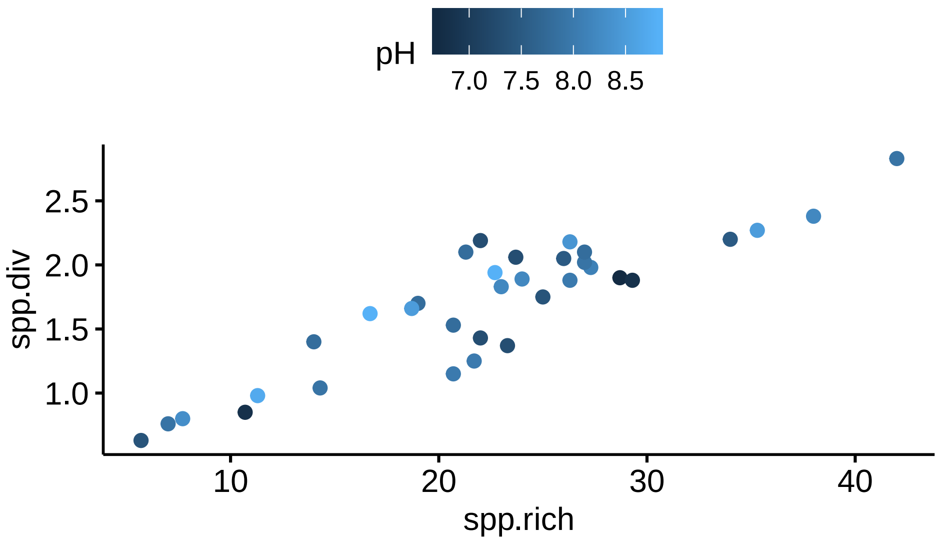
Now plot the scatter plot. Note that the syntax for ggpubr requires that variables be contained within quotes.

```
ggscatter(data = med98, y = "spp.div",x = "spp.rich")
```



We can color-code the points by their pH

```
ggscatter(data = med98, y = "spp.div",x = "spp.rich", color = "pH")
```



End optional section

10.9 Loading .csv files using RStudio

Frequently in code I will have things written up to load data using the **read.csv()** command. However, there is a point-and-click way of loading spreadsheet data into RStudio too.

There's on pane in RStudio that doesn't get used much by basic R users, the "Environment, History, Connections, Build, Git" pane (I think it might not have "Git" on it if you don't have certain packages loaded).

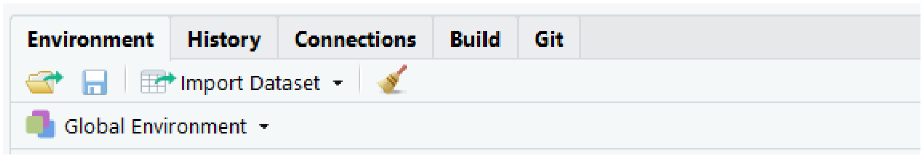


Figure 10.4: A list of files stored on GitHub.

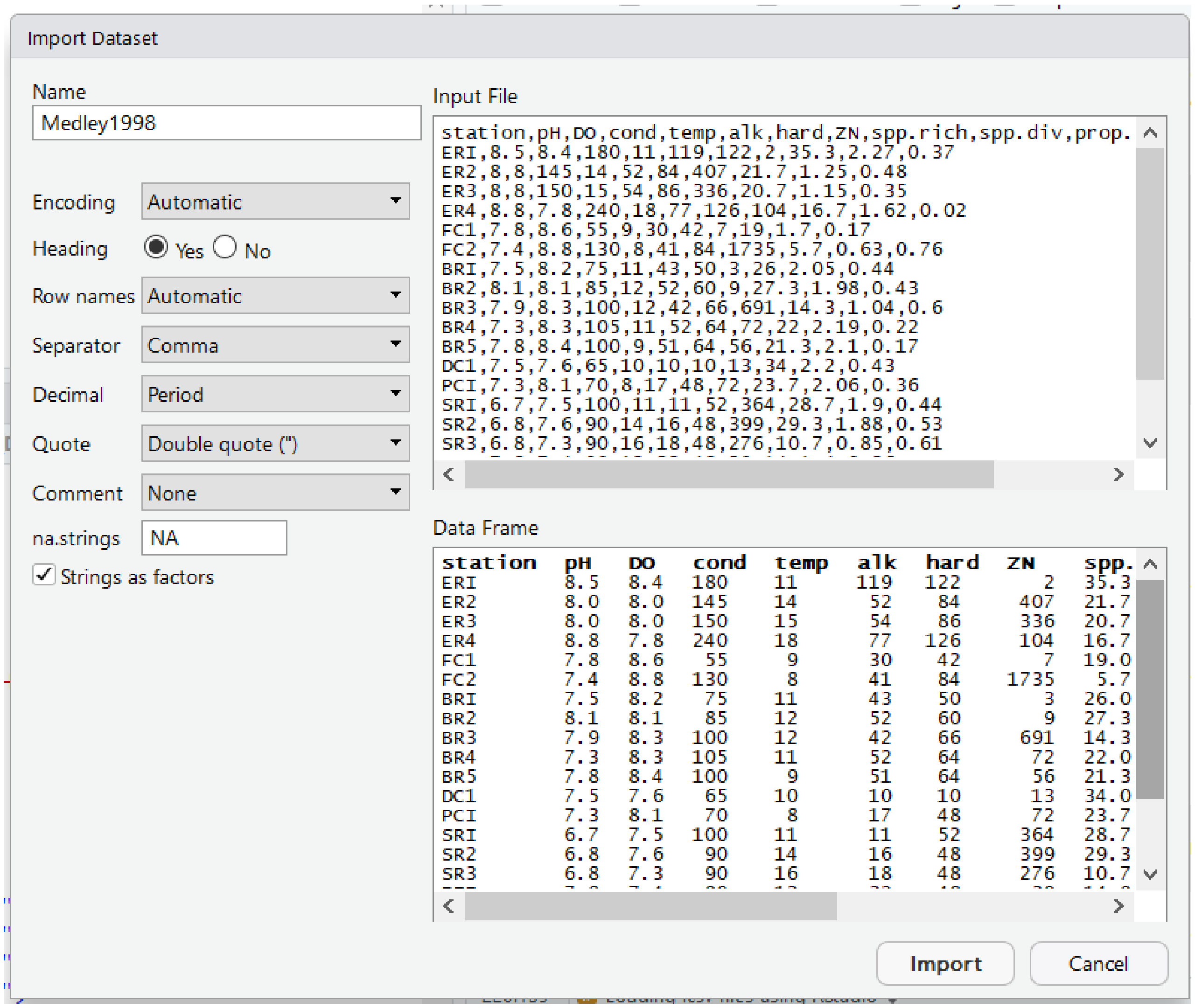
If you click on the spreadsheet-looking icon “Import Dataset” and select “From text (base)” you can navigate to where your .csv file is located and select it. A preview window will then pop up which will show you the raw (which should look like what you originally down loaded) and a preview of how RStudio will format the data. (If the preview doesn’t look right you can change some of the option in the drop down menus to see if things line up.)

When you click “Import” RStudio will execute some code in the console (eg “Medley1998 <- read.csv(”~/Medley1998.csv“)”) to load the data and then call the command **View()** to generate preview of the data in a new tab in the script view. (Note that this View panel only lets you look at the data; you can’t edit it.)

10.10 Challenge

.csv files are the most common format for sharing data in R. “csv” stands for “comma seperate volume”, and you will note that each value on a line is seperated by a comma (some things with computers do make sense on the first try!).

Sometimes you will encounter .txt files which separate data other ways, such as spaces, tabs, or by lining up everything explicitly in rows. On the wildlifeR GitHub directory we used before (https://github.com/brouwern/wildlifeR/tree/master/inst/extdata) these is a file “Medley1998.txt”. Download this file and load it using RStudio’s Import Dataset function. See if RStudio recognizes that its not .csv.



Name: Medley1998

Encoding: Automatic

Heading: [✓ Yes] No

Row names: Automatic

Separator: Comma

Decimal: Period

Quote: Double quote (")

Comment: None

na.strings: NA

[✓] Strings as factors

Input File

```
station,pH,DO,cond,temp,alk,hard,ZN,spp.rich,spp.div,prop.
ERI,8.5,8.4,180,11,119,122,2,35.3,2.27,0.37
ER2,8,8,145,14,52,84,407,21.7,1.25,0.48
ER3,8,8,150,15,54,86,336,20.7,1.15,0.35
ER4,8.8,7.8,240,18,77,126,104,16.7,1.62,0.02
FC1,7.8,8.6,55,9,30,42,7,19,1.7,0.17
FC2,7.4,8.8,130,8,41,84,1735,5.7,0.63,0.76
BRI,7.5,8.2,75,11,43,50,3,26,2.05,0.44
BR2,8.1,8.1,85,12,52,60,9,27.3,1.98,0.43
BR3,7.9,8.3,100,12,42,66,691,14.3,1.04,0.6
BR4,7.3,8.3,105,11,52,64,72,22,2.19,0.22
BR5,7.8,8.4,100,9,51,64,56,21.3,2.1,0.17
DC1,7.5,7.6,65,10,10,10,13,34,2.2,0.43
PCI,7.3,8.1,70,8,17,48,72,23.7,2.06,0.36
SRI,6.7,7.5,100,11,11,52,364,28.7,1.9,0.44
SR2,6.8,7.6,90,14,16,48,399,29.3,1.88,0.53
SR3,6.8,7.3,90,16,18,48,276,10.7,0.85,0.61
```

Data Frame

```
station  pH   DO   cond  temp  alk  hard   ZN  spp.
ERI      8.5  8.4  180   11    119  122     2  35.3
ER2      8.0  8.0  145   14     52   84   407  21.7
ER3      8.0  8.0  150   15     54   86   336  20.7
ER4      8.8  7.8  240   18     77  126   104  16.7
FC1      7.8  8.6   55    9     30   42     7  19.0
FC2      7.4  8.8  130    8     41   84  1735   5.7
BRI      7.5  8.2   75   11     43   50     3  26.0
BR2      8.1  8.1   85   12     52   60     9  27.3
BR3      7.9  8.3  100   12     42   66   691  14.3
BR4      7.3  8.3  105   11     52   64    72  22.0
BR5      7.8  8.4  100    9     51   64    56  21.3
DC1      7.5  7.6   65   10     10   10    13  34.0
PCI      7.3  8.1   70    8     17   48    72  23.7
SRI      6.7  7.5  100   11     11   52   364  28.7
SR2      6.8  7.6   90   14     16   48   399  29.3
SR3      6.8  7.3   90   16     18   48   276  10.7
```

Figure 10.5: A list of files stored on GitHub.

Chapter 11

Loading Excel spreadsheets into RStudio

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter: lobrowR

THIS CHAPTER HAS NOT BEEN WRITTEN

re-save as .csv and load

load directly

In this walk through we first re-save this data in an R-compatible format, a "csv" file, called "Lab1_data_PA_eagles.csv".

11.0.1 Prepping data in Excel

11.0.1.1 Save data to your R working directory (WD)

Find the file https://github.com/brouwern/wildlifeR/tree/master/inst/extdata

Save the file "Lab1_data_PA_eagles.xlsx" to your computers desktop. Today we will be using this as the "working directory"

11.0.1.2 Re-Save The Excel file as a "csv" file

In Excel, follow these steps

- "File"
- "Save As"
- "Browse"
- Select the working directory (your desktop)
- Select "Save as type"
- Select "CSV (Comma delimited)"
- Click "Save"

The data is now in a format that can be loaded into R.

11.1 Preparing a file for loading into R

Things work best when your Excel file is “clean” & only has exactly what you want in it. Any extra, accidental typing can cause problems or make things confusing. A good practice is to always highlight cells to the right of and below your data, right click & select “Delete”. This will remove any accidental typing that occurred. Do this to the cells below your data also.

11.2 Reload data

Reload data; be sure to include the “csv” at the end. Use this code “eaglesPA <- read.csv(file =”eaglesPA.xlsx“)”. NOTE: I changed the name of the file to include “_w_2_states” so that I wouldn’t overwrite the original file. Don’t use this code unless you changed the file name to the exact same thing

```
#Use this code, w/o the "#" in front of it
# eaglesPA <- read.csv(file = "eaglesPA.xlsx")

#NOTE: I changed the name of the file to include "_w_2_states" so that I wouldn't overwrite the origina
#eaglesPA <- read.csv(file = "./data/Lab1_data_PA_eaglesPA_w_2_states.csv")
```

Type ls() to see what is now in your workspace

```
ls()
```

```
##  [1] "crabs"           "eagle.df"        "eagles"          "eaglesWV.url"
##  [5] "eaglesWV.url_2" "eaglesWV_2"     "iris"            "med98"
##  [9] "msleep"          "my.abc"          "my.mean"         "x"
## [13] "year"
```

Look at the re-loaded eaglesPA data object

```
summary(eaglesPA)
dim(eaglesPA)
head(eaglesPA)
tail(eaglesPA)
```

Part III

Plotting data in R with ggplot2 & friends

In this section we'll focus on basic plotting skills. The first chapter in this section is a review of the skills needed to get some data into R from a package. Subsequent chapters will develop plotting skills with a focus on using ggplot2.

We'll briefly go over all aspects of **ggplot2** and its varients. This will orient you to the different things you are likely to see other R users use. For most of this book, however, we'll focus on a package which creates a "wrapper" for ggplot2 called ggpubr which speeds up the process for beginners of making publication-ready graphs. Even if you are a ggplot2 pro you should know about ggpubr for making quick graphs, and for teaching others about ggplot2.

Chapter 12

Review: Loading & Examining Data in R

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter: lobrowR

12.1 Introduction

This exercise reviews the basics of loading data from a package into R. These basic skills are needed for the further tasks of plotting that will be built upon in this section. If you are familiar with R you can probably skim this or skip it altogether. If you are brand new then this is an excellent review of the material covered so far in this book.

12.1.1 Learning Goals & Outcomes

To review the basics of loading packages and data.

12.1.2 Functions & Arguements

- data
- library
- ls
- dim
- names
- head
- tail
- is
- summary
- install.packages

12.1.3 Packages

- MASS
- cowplot

12.1.4 Outline

- Intro
- Fisher's Iris data
- Loading data: iris
- The easy way: from base R w/ data(iris)
- Loading packages in base R: MASS
- command: library(MASS)
- Loading packages from CRAN: cowplot

12.2 Example data for plotting: Fisher's Irises

- Dataset made popular by R.A. Fisher
- Frequently used to explain/test stats procedures
- See ?iris for more details
- See also https://en.wikipedia.org/wiki/Iris_flower_data_set for more info.

12.3 Loading data into R the easy way: pre-made data in an R "Package"

- Getting data into R (or SAS, or ArcGIS…) can be a pain!
- R comes with many datasets that are pre-loaded into it
- There are also many stat. techniques that can easily be added to R
- These are contained in "packages"

12.3.1 Load data that is already in the "base" distribution of R

Fisher's iris data comes automatically with R. You can load it into R's memory using the command "data()"

```
data(iris) #Load the iris data
```

12.3.2 Look at the iris data

We'll look at the iris data using some commands like ls(), dim(), and names().

You can check that it was loaded using the **ls()** command ("list").

```
ls()
```

You can get info about the nature of the dataframe using commands like **dim()**

```
dim(iris)
```

This tells us that the iris data is essentially a spreadsheet that has 150 rows and 5 columns.

We can get the column names with names()

```r
names(iris)
```

- Note that the first letter of each word is capitalized.

- What are the implications of this?

The top of the data and the bottom of the data can be checked with head() and tail() commands

```r
head(iris) #top of dataframe

tail(iris) #bottom of dataframe
```

Another common R command is **is()**, which tells you what something is in R land.

```r
is(iris)
```

- R might spew a lot of things out at you when you use **is()**
- usually the 1st item is most important.

- Here, it tells us that the "object" called "iris" in your workspace is 1st and foremost a "data.frame"
- A dataframe is essentially a spreadsheet of data loaded into R.

You can get basic info about the data themselves using commands like **summary()**.

```r
summary(iris)
```

- Which variables are numeric?
- Which variables are categories/groups (aka "factors")?

If you wanted info on just 1 column, you would tell R to isolate that column like this, using a dollar sign ($).

```r
summary(iris$Sepal.Width)
```

That is, that name of the dataframe, a dollar sign ($), and the name of the column.

What happens when you don't capitalize something? Try these intentional mistakes (but remove the "#" from in front of each one):

```r
#all lower case
summary(iris$sepal.width) # this won't work

#just "s" in "sepal" lower case
summary(iris$sepal.Width)  #this won't work either

#or what if you capitalize "i" in "Iris"?
summary(Iris$Sepal.Width) #won't work either
```

The first two error messages are not very informative; the 3rd one ("Error in summary(Iris$Sepal.Width) : object 'Iris' not found") does make a little sense.

12.4 Load data that is in another R package

12.4.1 Packages that come with R

- Many scientists develop software for R, and they often include datasets to demonstrate how the software works.

- Some of this software, called a "package" comes with R already and just needs to be loaded.

- This is done with the **library()** command.
- The **MASS** package comes with R when you download it and has many useful functions and interesting datasets.

```
library(MASS) #Load the MASS package
```

MASS contains a dataset called called "mammals"

```
data(mammals)
```

You can confirm that the mammals data is in your workspace using **ls()**

```
ls()
```

You should now have both the "iris"" and the "mammals"" data in your R "workspace.""

What is in the mammals dataset? Datasets actually usually have useful help files. Access help using the **?** function.

The help screen will pop up either within RStudio, or possibly in your web browser. It tells us that mammals is

> "A data frame with average brain and body weights for 62 species of land mammals."

Since this is someone else's data, the authors of the MASS package need to provide proper citation. At the bottom we can see that these data come from the paper:

> Allison, T. and Cicchetti, D. V. (1976) Sleep in mammals: ecological and constitutional correlates. Science 194: 732-734.

We can learn about the mammals data using the usual commands

```
dim(mammals)
names(mammals)
head(mammals)
tail(mammals)
summary(mammals)
```

12.5 Load Data From A package On CRAN

Most packages don't come with R when you download it but are stored in a central site called CRAN. We'll load data from the **cowplot** package.

12.5.1 Loading packages using R-Studio

RStudio makes it easy to find and load packages. Follow these instructions.

- In the panel of RStudio that has the tabs "Plots", "Packages","Help", "Viewer" click on "Packages""
- On the next line it says "Install" and "Update". Click on "Install"
- A window will pop up. In the white field in the middle of the window under "Packages" type the name of the package you want.
- RStudio will automatically bring up potential packages as you type.
- Finish typing "cowplot" or click on the name.
- Click on the "Install" button.
- In the source viewer some misc. test should show up. Most of the time this works. If it doesn't, talk to the professor!

If an R packages doesn't load properly, it could be for several reasons.

1. Your internet connection might be having problems.

2. The website where the package is stored might be down for maintenance.

3. The version of are you are using is probably newer than the version of R used to make the package. This is a real pain - ask for help from an expert R user if think you have this problem.

12.6 Loading packages directly using code

You can also use the **install.packages()** command to try to load the package.

```
install.packages("cowplot")
```

12.7 Troubleshooting Package Downloads

12.7.1 What if you tell R to install a package you already have downloaded?

If you already have the package downloaded to your computer then a window will pop up asking you if you want to restart your computer. Normally this isn't necessary; just click "no". You might see a "warning" message pop up in the console such as "Warning in install.packages: package 'cowplot' is in use and will not be installed". This isn't a problem for basic R work. If you are doing serious work (e.g. for a publication) you should restart R.

12.7.2 What if I can't get a package I need loaded?

- Talk to someone who is good w/R (eg, your professor)
- Google something like "how to install R package" for general info
- Google something like "problem loading R package"
- Copy and paste any error message you might be getting into Google and see if anyone has written about this problem

See above for reasons why a package might not load properly the 1st time you try.

12.7.3 Finding R help with Google

There's lots of info about R on the web, and if you have a problem, then someone else has probably had it before and perhaps written something about it.

The website **stackoverflow.com** has lots of info about R. However, many people who use it are hard-core programmers, who can come across as jerks sometimes when they answer questions if you don't follow the rules and protocols of stackoverflow.

Chapter 13

Plotting Continous Data in R With ggplot2

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter: lobrowR

13.1 Introduction

- We're going to plot some data using the qplot() command
- We'll need to have 2 packages loaded
- ggplot2, which has the function qplot()
- cowplot, which provides some nice defaults
- We'll use the iris dataset that comes with R

13.1.1 Learning goals & objectives

Introduce the qplot command from the ggplot2 package

13.1.2 Functions & Arguements

13.1.3 Packages

- ggplot2
- cowplot

13.1.4 Outline

13.2 Introduction to ggplot using qplot

Load the iris data

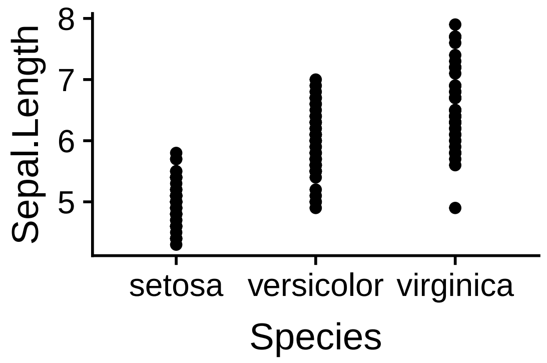
```
data(iris)
```

Load the ggplot2 and cowplot packages

13.3 A basic plot in ggplot using qplot()

Unless you tell it otherwise, qplot plots dots.

```
qplot(y = Sepal.Length,
      x = Species,
        data = iris)
```



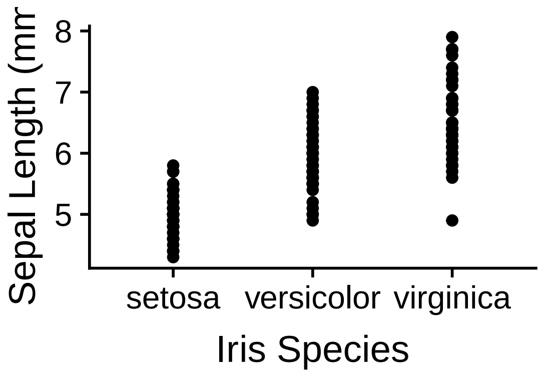
Chapter 14

Box plot with labels

- R will usually generate labels for the x and y axes based on the command. * These can be changed by adding another command after the qplot() command
- Add The command + **xlab("...")** sets the labels for the x-axis, + **ylab("...")** for the y axis.
- Text for the labels goes in quotes (ie, "Iris species").
- The use of the "+" is different than for most other R packages
- Forgetting the quotes will cause the code to fail.

- Note that units (mm) are included for the y axis.

```
qplot(y = Sepal.Length,
      x = Species,
        data = iris) +         # note the "+"
  xlab("Iris Species") +       # label for x axis
  ylab ("Sepal Length (mm)" )  # label for y axis
```



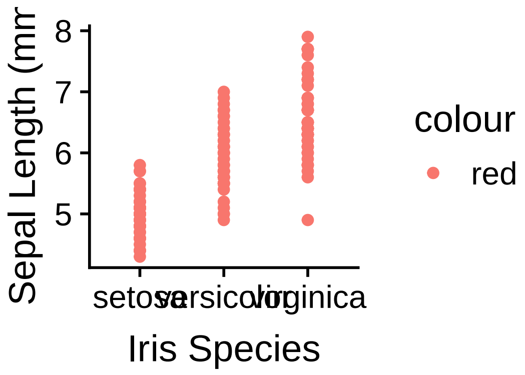
14.1 Changing colors in R plots

14.1.1 Changing colors in R plots part 1

- If we wanted we could change the color of the dots using the argument **"col ="**. This code can be used to change the color of most types of plots in R.

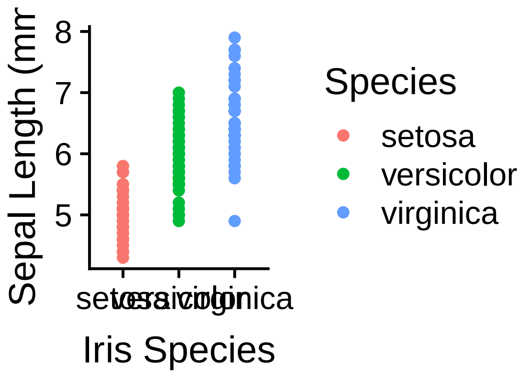
- This doesn't increase the information content of the figure but maybe makes it nicer to look at.

```
qplot(y = Sepal.Length,
      x = Species,
        data = iris,
      color = "red") +
  xlab("Iris Species") +
  ylab ("Sepal Length (mm)" )
```



14.1.2 Changing colors in R plots part 2

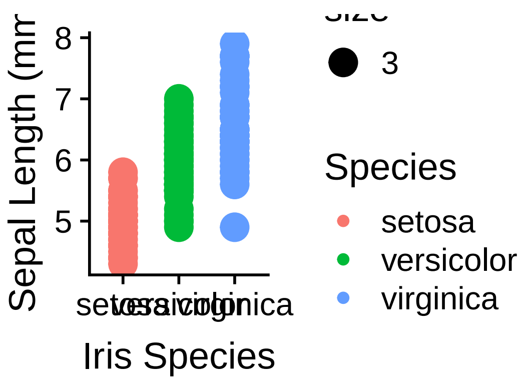
```
#dopt w/color changes
qplot(y = Sepal.Length,
      x = Species,
        data = iris,
      color = Species) +
  xlab("Iris Species") +
  ylab ("Sepal Length (mm)")
```



14.2 Tweaking plots: changing the point size

Run the code below, Can you see what changed?

```
#dopt w/color changes
qplot(y = Sepal.Length,
      x = Species,
        data = iris,
      color = Species,
      size = 3) +
  xlab("Iris Species") +
  ylab ("Sepal Length (mm)")
```

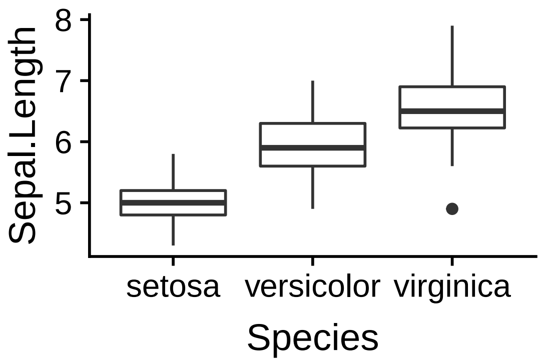


14.3 Boxplot with qplot

14.3.1 Basic boxplot with qplot

- note use of arguement **"geom = ..."**

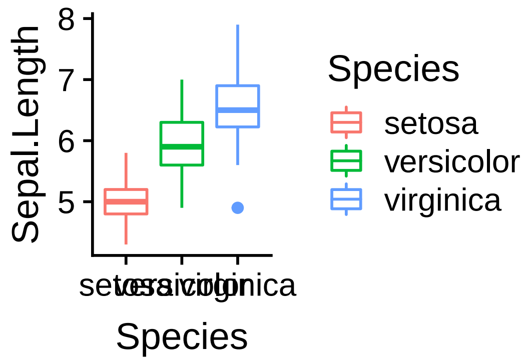
```
qplot(y = Sepal.Length,
      x = Species,
        data = iris,
      geom = "boxplot")
```



14.4 Basic boxplot with colors

- same as before, using "color ="

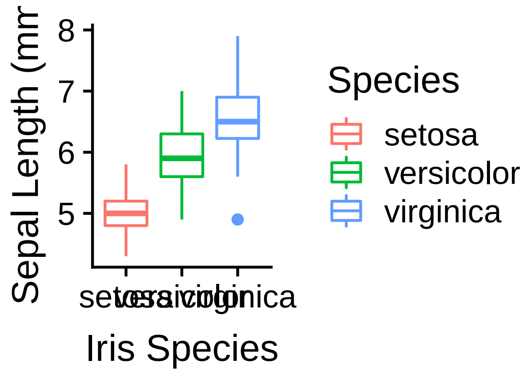
```
#dopt w/color changes
qplot(y = Sepal.Length,
      x = Species,
        data = iris,
      geom = "boxplot",
      color = Species)
```



14.5 Basic boxplot lables

- now use + xlab() and + ylab()

```
qplot(y = Sepal.Length,
      x = Species,
        data = iris,
      geom = "boxplot",
      color = Species) +
  xlab("Iris Species") +
  ylab ("Sepal Length (mm)")
```



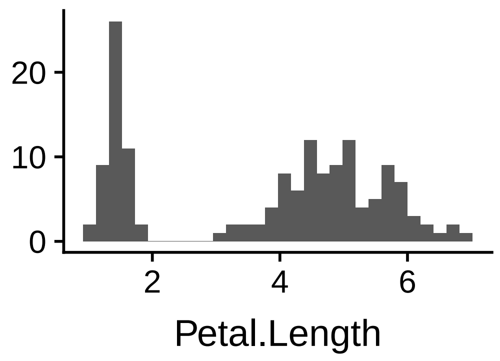
14.6 Histograms using qplot

- made with geom = “histogram” arguement
- very very easy to make in R with ggplot
- very very very hard to make in Excel
- You should make them all the time for you data!

14.6.1 Histograms of iris data

- This code makes a histogram of one of the iris species' Petal.Length.
- Note that you don't have "y =" or "x =" for a histogram!

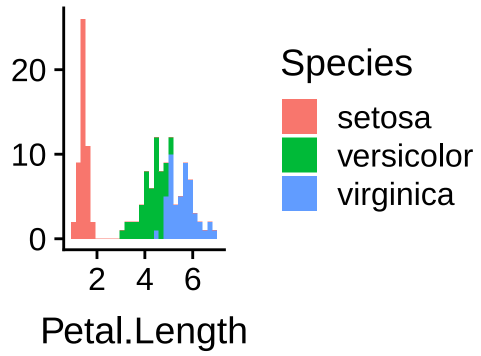
```
qplot(Petal.Length,
      data = iris)
```



14.6.2 Histogram with colors

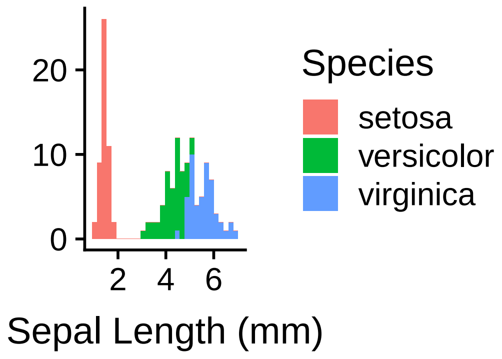
What does this show?

```
qplot(Petal.Length,
      data = iris,
      fill = Species)
```



14.6.3 Histogram with axes labels

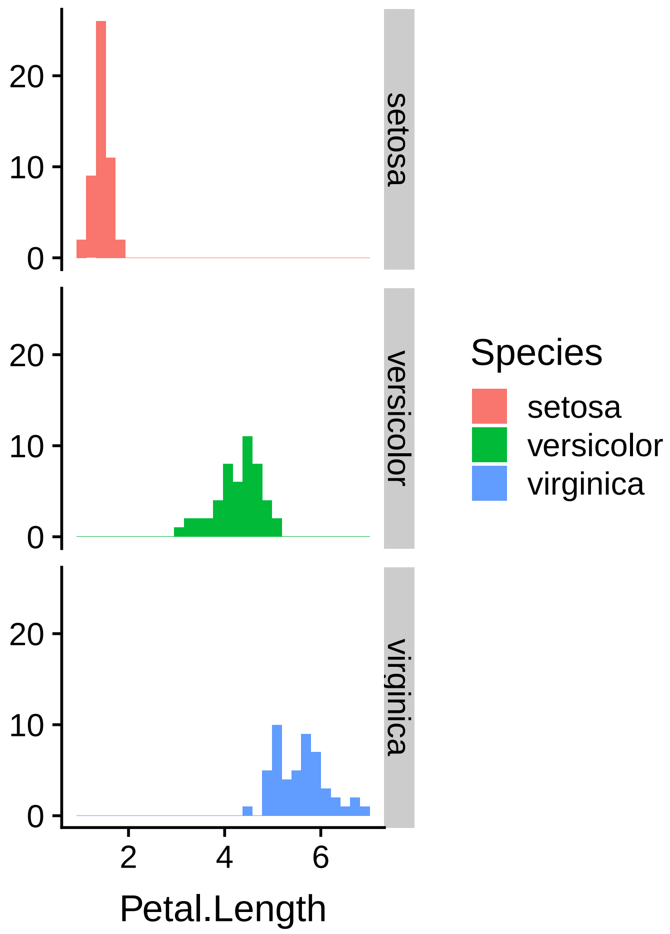
```
qplot(Petal.Length,
      data = iris,
      fill = Species) +
  xlab ("Sepal Length (mm)")
```



14.6.4 Multiple histograms: "Facets"

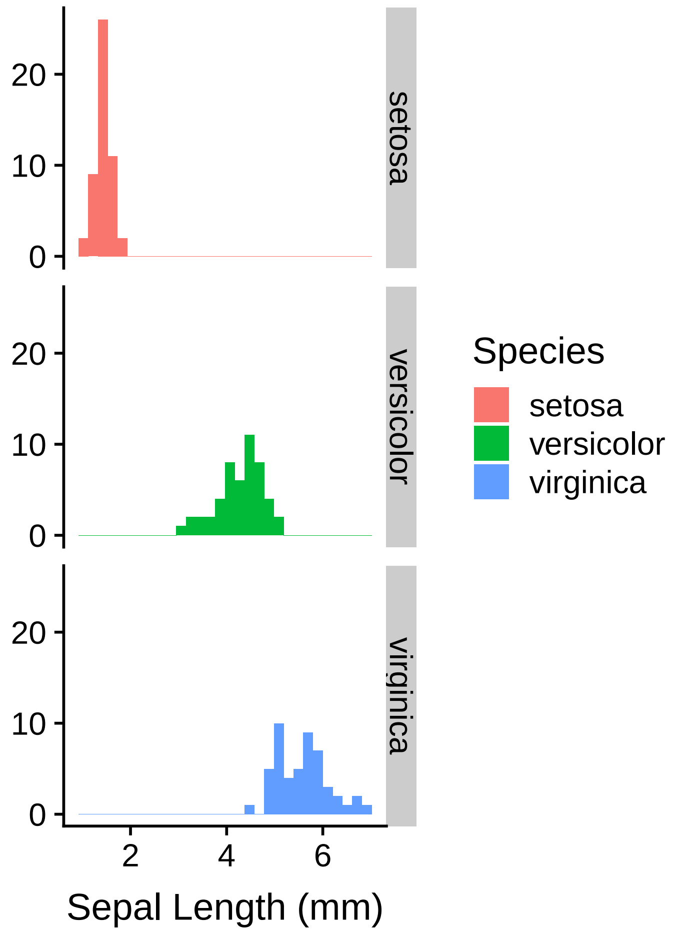
What does this show?

```
qplot(Petal.Length,
      data = iris,
      fill = Species,
      facets = Species ~.)
```



Add a label to x-axis

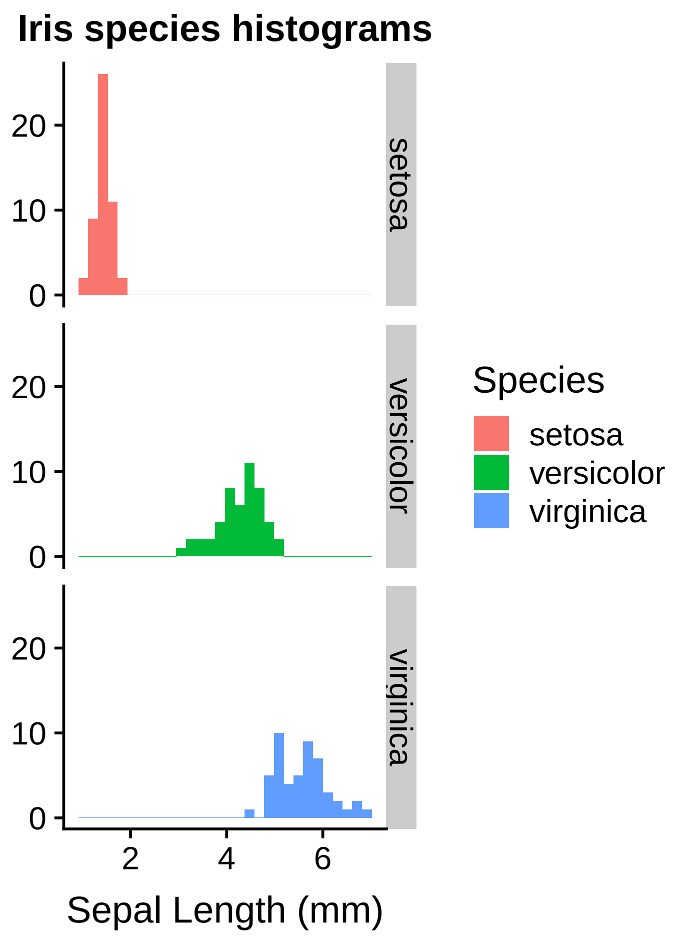
```
qplot(Petal.Length,
      data = iris,
      fill = Species,
      facets = Species ~.) +
  xlab("Sepal Length (mm)")
```



14.7 Modifying histograms: titles with the main = argument

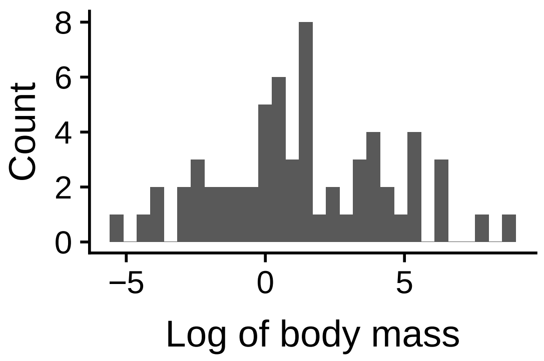
- Titles are good for your own personal use but actually are almost never used in figures published in papers and books.
- We can add a title like this using the arguement “main =”

```
qplot(Petal.Length,
      data = iris,
      fill = Species,
      main = "Iris species histograms",
      facets = Species ~.) +
  xlab ("Sepal Length (mm)")
```



14.8 Challenge: Make a histogram of the mammals data

- Load the MASS library using library(MASS)
- Load the mammals data using data(mammals)
- Make a histogram of the log of body mass
- Add labels using xlab() and ylab()



Chapter 15

Scatterplots in R Using qplot()

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter: lobrowR

15.1 Introduction

We'll show how to make **scatterplots** using the "quick plot" function (**qplot**) from ggplot2. ggplot2::qplot and ggpubr both offer simplified plotting using tools from ggplot2's toolkit. ggpubr implements a different syntax, though, while qplot use more standard ggplot2 idioms and so is a good way to get a sense for the full power of ggploting.

Learning goals & outcomes

- Make scatter plots :)

Functions & Arguements

- library
- names
- qplot
- data
- dim
- head
- summary
- factor
- log

Packages

- ggplot2
- cowplot

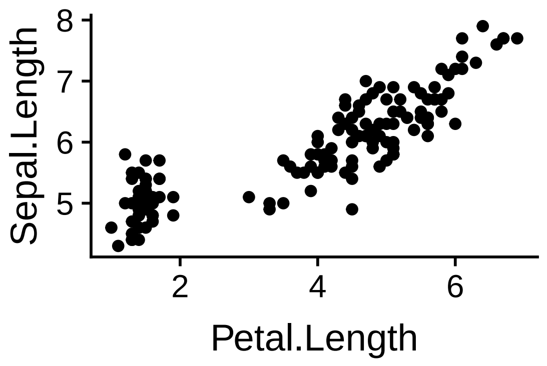


Figure 15.1: Sepals vs. Petals

Potential hangups

There are many ways to make plots in R: ggplot, qplot, ggpubr, plot; and I'm leaving a few out. Moving between them is meant to give you a sense for the different tools so you can recognize them "in the wild" on the internet and in books. We'll transition to focusing on ggpubr soon.

15.2 Scatterplots: 2 Continuous Variables

In this lab we'll explore how to make scatterplots using the qplot() function in ggplot2.

15.2.1 R Preliminaries

- We'll use the qplot() function in the *ggplot* package
- The *cowplot* package provides nice deafults for ggplot IMHO

15.2.2 Scatterplot of Iris data

- Let's make a scatter plot, where we plot two continous, numeric variables against each other
- that is, both x and y variables are numbers; not categories

I've forgotten the names of all the iris variables, so I'll use the **names()** command to see what they are

```
names(iris)
```

I'll plot the sepals against the petals

```
qplot(y = Sepal.Length,
      x = Petal.Length,
      data = iris)
```

15.2.3 Scatter plot of mammal brain data

Let's look at another dataset

15.2.3.1 Preliminaries

Get the data from the ggplot2 package

```
data(msleep)
```

15.2.3.2 Look at the data

```
dim(msleep) #How much data is there?

head(msleep) #What does the data look like

summary(msleep) #Summary of the data
```

There are a number of "categorical" varibles in this dataset

- genus
- vore = carnivore, omnivore et
- order = taxonomic order
- conservation = conservation status (endangered, etc)

For some reason they don't load as "factor" variables (better known as categorical or grouping variables, but called "Factors" in R-land)

We can make these factors using the factor() command

```
msleep$vore <- factor(msleep$vore)
```

Now see what happens when you call summary()

```
summary(msleep)
```

Do the same for "order""

```
msleep$order <- factor(msleep$order)

summary(msleep)
```

And "conservation"

```
msleep$conservation <- factor(msleep$conservation)

summary(msleep)
```

15.2.4 Make a basic scatterplot

```
qplot(y = sleep_total,
      x = brainwt,
      data = msleep)
```

That looks really really ugly. It will work better if we "log transform the axes"

```
qplot(y = log(sleep_rem),
      x = log(brainwt),
      data = msleep)
```

Things get logged all the time in stats. We'll talk more about that later.

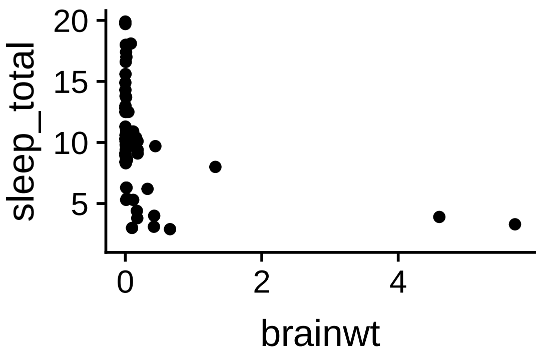


Figure 15.2: Mammal sleep, raw data

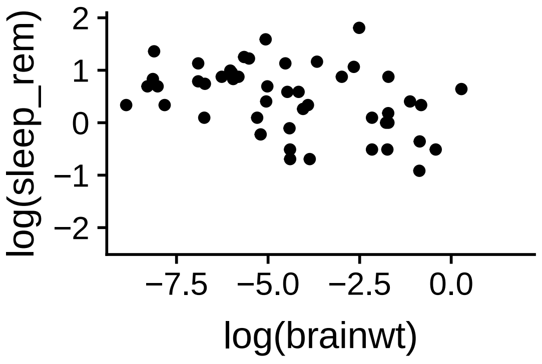


Figure 15.3: Mammal sleep, logged data

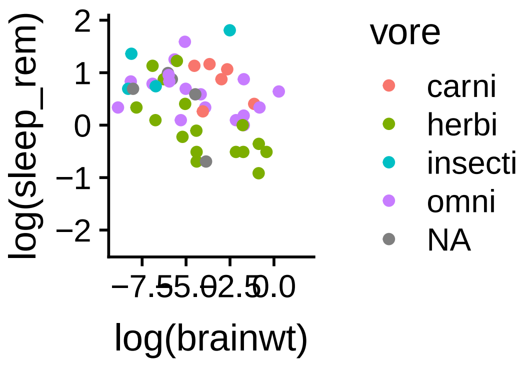


Figure 15.4: Add colors with color =

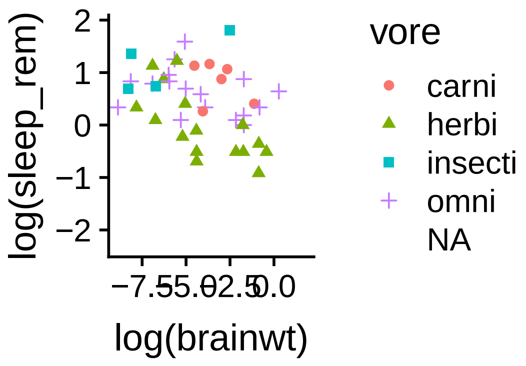


Figure 15.5: Add shapes with shape =

15.2.5 Add color coding to scatterplot

```
qplot(y = log(sleep_rem),
      x = log(brainwt),
      data = msleep,
      color = vore)
```

15.2.6 Add color & shape coding to scatterplot

```
qplot(y = log(sleep_rem),
      x = log(brainwt),
      data = msleep,
      color = vore,
      shape = vore)
```

15.2.7 Put diffetrent "vores" in seperate panels

- Seperate panels can be made using the "facet" arguement withing qplot

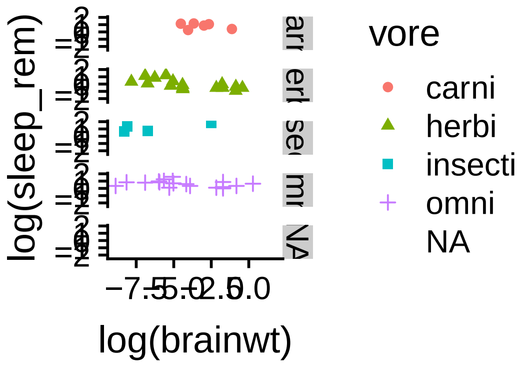


Figure 15.6: Split into different panels w/ facets =

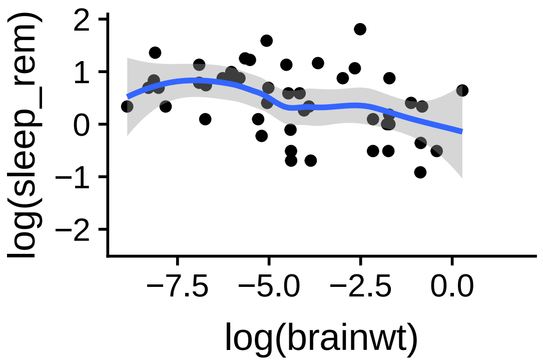


Figure 15.7: (#fig:last.chnk.sxn3.ch3)Add trendline with + geom_smooth()

```
qplot(y = log(sleep_rem),
      x = log(brainwt),
      data = msleep,
      color = vore,
      shape = vore,
      facets = vore ~ .)
```

15.2.8 Add a "trend line"" to a scatterplot

- Add the geom_smooth() function after the initial qplot() command
- This works best if we remove the "color = vore" command, but you can see what happens if you leave it

```
qplot(y = log(sleep_rem),
      x = log(brainwt),
      data = msleep) +
  geom_smooth()
```

15.2.9 Challenge: Modify mammal brain code

Modify the mamal bran code to do the following things

- Change the axes labels (eg "+ ylab('y axis')")
- Add a title (eg " + ggtitle('…')")
- Use names(msleep) to see what other varibles are in the dataset
- Use summary(msleep) to whether they are continous or categorical
- Pick another continous variable and plot it instead of sleep_total
- Try this with and without logging using the log() command

Part IV

Data analysis: A First Encounter

In this section we will tackle a typical data analysis problem: determining if two groups, such as organisms or drug treatments, an be considered statistically different from each other. We will use data from a paper titled “Sperm competition and the evolution of precopulatory weapons: Increasing male density promotes sperm competition and reduces selection on arm strength in a chorusing frog” by Buzatto et al (2015).

The end goal is to compare the size of the arms on female and male frogs. First, though, we will get to know the data by calculating summary statistics and making exploratory graphs. We will then carry out a t-test and grapple with with the meaning and interpretation of the output Finally, we’ll explore how best to plot the output of a t-test.

Section outline:

1. Data exploration with summary statistics
2. Graphical data exploration with boxplots
3. Plotting means and measures of variation and precision
4. T-tests
5. Plotting the output of a t-test

Chapter 16

Data Analysis Encounter: Summary Statistics

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter: lobrowR

16.1 Introduction

16.1.1 Goals and objectives

Get to know the **frogarms** dataset and learn how to calculate summary statistics using basic R functions (eg **summary()**) and also with the handy tools in the **dplyr**.

16.1.2 Outline

1. Load packages and data
2. Subset unique dataset
3. Calculate summary stats on columns
4. Use dplyr
5. Calculate summary stats by groups using dplyr

16.1.3 Packages

- devtools
- wildlifeR (from GitHub)
- dplyr

16.1.4 Functions

- devtools::install_github
- wildlifeR::make_my_data2L
- dim, nrow, ncol

- head, tail
- names
- summary
- median
- min, max, range
- var, sd
- nrow, length (for sample size)
- dplyr::summarise, dplyr::summarize
- group_by
- dplyr::n

16.2 Preliminaries

First, we need to install the necessary packages. The data are in a package stored on GitHub called **wildlifeR**. The **devtools** package is needed for downloading from Github. We'll also use **dplyr** for grouping data and calculating summary statistics.

16.2.1 Load packages

You might have to install or re-install wildlifeR using **install.packages()**. **If you have done this recently you can skip this step.**

```
library(devtools)

install_github("brouwern/wildlifeR") #Note that text is quoted " "
```

Recall that *downloading* a package and actually *loading* it into R's active memory are different things. To actually use the package you need to use the **library()** command to load it into memory.

16.2.2 Load data

The data we'll be using is in a dataset called "frogarms" in the wildlifeR package.

```
data(frogarms) #[_]
```

You can find out information about these data using the ? command. Note that there are no parentheses required for this ("?(frogarms)" is wrong)

16.2.3 Subset your data

In this lesson we'll be primariy working with a personalized subset of the data. This will allow us to

1. See the effects of sample size by comparing the larger frogarms data to your subset
2. See the effects of random variation on things like p-values

The worksheet that accompanies this chapter is meant to facilitate these comparisons between the full (frogarms) and sub datasets, and also between you can classmates. The code that follows is focused on working with the subset we will generate below, but the same commands should also be run on the full frogarms dataset.

The function **make_my_data2L()** in the wildlfieR will extract out a random subset of the data. Change "my.code" to your school email address, minus the "@ pitt.edu" or whatever your affiliation is.

```
my.frogs <- frogarms # [_]
my.frogs <- make_my_data2L(dat = frogarms,
                           my.code = "nlb24", # <=  change this!
                           cat.var = "sex",
                           n.sample = 20,
                           with.rep = FALSE)
```

n.sample is set to 20. This is set up to extract 20 unique individuals of each sex (20 male, 20 female). Check that you dataframe is 2*20 = 40 rows using the dim() command.

```
dim(my.frogs) # [_]
```

16.2.4 An aside on R functions (Optional) [O]

The following sections are optional. The 2sst task is easy to wrap your brain around (looking at the code behind a function); the 2nd is more advanced (debugging a function).

16.2.4.1 OPTIONAL (easy): Looking at the code behind a function

** This section is optional, but easy for beginners**

Functions in R are typically written using R code. To see the underlying code you can type the function name in the console but nothing else, then run the command

```
make_my_data2L
```

The formatting of the output in the console might look a bit goofy; you can adjusted the conlse size so it looks better. This code is fairly long but is mostly fairly basic R commands and is a single function. In contrast, many R functions call other functions within them..

You can't always see the underlying code for a function though. Try to look at the t.test function

```
t.test
```

16.2.5 OPTIONAL (intermediate): Function defaults

This section is optional

"Statistics is the science of defaults" (6 November 2012, https://andrewgelman.com)

See what happens when you run this code; note that there is **no** "my.code = …" bit in it (just to simplify things).

```
make_my_data2L(dat = frogarms, cat.var = "sex")
```

Why does this work? Check out the helpfile using **?make__my__data2L**, or look at the raw code as we just did. Notice that right after the function name is listed the following things

- dat
- my.code = "nlb24"
- cat.var
- n.sample = 20
- with.rep = FALSE

These are the **arguements** that **make_my_data2L()** takes. When an arugment name is followed by an "= …" that means a default has been set. If you call the function and don't specify what you want for a specific arguement, R checks for pre-specified **defaults** and uses those as needed. Note that the use of defaults can be problematic, since you might use a deafult you didn't intend to. Most defaults are sensible, and essential things like dataframes that need to be supplied rarely have defaults.

16.2.6 OPTIONAL (Advanced): Debugging a function

This section is optional and not relevant for beginners

When a function is not working, or you want to understand how it works, you can debug it. First, tell R that the next time you run the function you want to debug it

```
debugonce(make_my_data2L)
```

Then run the function

```
make_my_data2L()
```

This will create a new tab in **debugger mode**. Every time you press "enter" you will step through the code to the next full line of R code (note that a line of functional R code can span more than one line of code in a file or when rendered on a screen). When you get to the end of the function you will exit the debugger and go back to normal R mode.

If you want to interact with the function while its debugging you can type directly in the console. For example, trying running the ls() command after every few lines of code to see what happens. You can call dim(), summary(), is() etc on anything you find.

END OPTIONAL SECTION

16.3 Getting to know your personalized dataframe

Let's get to know your personalized subset of the data.

16.3.1 Dataframe dimension

The following commands tell you the row x column dimension, number of rows, and number of columns.

```
dim(my.frogs)  #[_]
nrow(my.frogs) #[_]
ncol(my.frogs) #[_]
```

16.3.2 Optional: Accessing elements of objects

The following section is optional.

The commands dim(), nrow and ncol all generate objects that display information on the dimension of a dataframe. dim() produces and object that is a "vector" that is 1 row x 2 elements in size. We can select these individual elements using square brackets "[…]"

The full dim() output

```
dim(my.frogs)
```

The 1st element of the dim() output is accessed by appending "[1]" to the very end of the line of code

```
dim(my.frogs)[1] #"[1]" goes outside the ")"
```

The 2nd element

```
dim(my.frogs)[2]
```

Both elements: use "[1:2]" with a ":".

```
dim(my.frogs)[1:2]
```

Another way to get both elements, using "c(...)"

```
dim(my.frogs)[c(1,2)]
```

What happens when you execute the command below command? Why?

```
dim(my.frogs)[c(2,1)]
```

If you want to know one reason why R can drive you crazy, run this code, without the "c" before "(1,2)"

```
dim(my.frogs)[(1,2)]
```

But then try this; same as above but with ":" instead of "," between the numbers.

```
dim(my.frogs)[(1:2)]
```

Yes, R is that picky.

End optional section

16.3.3 Dataframe preview

The **head()** and **tail()** commands give us short previews of the dataframe.

head() gives the top few rows

```
head(my.frogs) #Note: I've truncated the output that I've actually shown
```

The bottom few rows with **tail()**

```
tail(my.frogs)
```

names() just gives of the names of the columns. It is the same as **colnames()**.

```
names(my.frogs)
```

Again, if you want to know what the column names are, use the ? command

```
?my.frogs
```

16.4 Summary statistics

This section is review. If you are familar with R you can skip ahead

R is a giant calculator. There are commands for **mean**, **median**, **standard deviation** etc. The **summary()** command creates a handy summary, including the mean and median, of all columns in a dataframe.

16.4.1 Overall summary

Whole dataframe

```
summary(my.frogs)
```

We can look at just a single column by specifying it using the syntax “dataframe*column.nameswherethedataframeandcolumn*
(note that it prints it out horizontally, not vertically)

```
summary(my.frogs$mass)
```

We used the **make_my_data2L()** command to make a unique subset of the data. compare the mass values in your subset to the original data

```
summary(my.frogs$mass)
summary(frogarms$mass)
```

End review section

16.4.2 Optional: stacking things with rbind()

This section is optional

Handy trick: stack up summaries with **rbind()**, which stand for “row-bind”.

```
rbind(summary(my.frogs$mass),  #note the comma
      summary(frogarms$mass))
```

You can even flip them on their side like this

First, make an object with your summaries

```
my.summaries <- rbind(summary(my.frogs$mass),
                      summary(frogarms$mass))
```

Flip them with the **t()** command (“t” stands for “transpose”)

```
t(my.summaries)
```

End optional section

16.4.3 Individual summary stats

This section is review. If you are familar with R you can skip ahead

You can get individual summary statistics using various commands named after the statistic.

The mean of a column with **mean()**.

```
mean(my.frogs$mass)
```

The variance with **var()**.

```
var(my.frogs$mass)
```

Other include:

- median
- min, max, range
- var, sd
- nrow or length() (for sample size)

Note that **range()** returns 2 values in a **vector**

```
range(my.frogs$mass)
```

16.4.4 The standard error (SE) in R

Note that R doesn't return a very common statistic, the **standard error (SE)**. The SE is the standard deviation (SD) divided by the square root of the sample size. You can get the same size using the length() command.

You can therefore calculate the SE like this:

```
sd(my.frogs$mass)/sqrt(length(my.frogs$mass))
```

16.4.5 OPTIONAL: Find a package the calcualtes the SE [O]

This section is optional

In the following 2 optional sections you can

- try to find a package with an SE function
- try to write a function that calculates the SE for you

Since R lacks a an SE function many packages include it. For example, the **plotrix** package has a function **std.error()**. See if you can download the package, install it using **library()**, and use **std.error()**. See the help file for more info (?std.error).

Try to look at the underlying code either in the console or by running the debugger using debugonce().

16.4.6 OPTIONAL: Write your own SD function [O]

Write a function for calculating the SD

Here's a function that takes a single argument "dat_column"

```
#NOTE: this is optional
my_sd1 <- function(dat_column){
  sd(dat_column)/sqrt(length(dat_column))
}
```

To use it, you need to give it the dataframe and the column separated by a "$""

```
my_sd1(my.frogs$mass)
```

Here's a function that takes 2 arguments: the dataframe, and the name of the column Note that the name of the column needs to be in quotes

```
my_sd2 <- function(dat, column){
  sd(dat[,column])/sqrt(length(dat[,column]))
}
```

You can use the function like this:

```
my_sd2(my.frogs, "mass") #note the use of quotes "..."
```

Here's a fancier function that let's you specify how much to round off the results. I've set the default rounding to 3 digits.

```
my_sd3 <- function(dat, column, digits.round = 3){
  se <- sd(dat[,column])/sqrt(length(dat[,column]))
  round(se, digits = digits.round)
}
```

The function runs like this.

```
my_sd3(my.frogs, "mass")
```

Note that in all of functions as long as I give the function the **arguments** in the same order they are set up in the code that defines the function, I don't need to provide the agruement names. This save typing. Compare these results

```
my_sd3(my.frogs, "mass")
my_sd3(dat = my.frogs, column =  "mass")
my_sd3(column =  "mass", dat = my.frogs)
```

Now try this

```
my_sd3("mass", my.frogs)
```

Can you figure out what has happend with the last one?

End optional section

16.5 A 1st encounter with dplyr [__]

dplyr is a package that provides numerous functions for manipulating data. It is part of the expanding **tidyverse** of packages sponsored in large part by RStudio. Hadley Whickham is the primary achitect of the tidyverse; he wrote many of the first packages in this framework and laidout the overall conceptual basis that other package authors follow.

For more on dplyr see

- https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html
- https://dplyr.tidyverse.org/
- http://genomicsclass.github.io/book/pages/dplyr_tutorial.html

We will use 2 **dplyr** handy functions

- summarize() / summarise()
- group_by()

dplyr can use a syntax that involves "**pipes**". This is a relatively recent innovation in R coding. You can string together R commands using the **pipe function**, %>%.

Note that the pipe function actually is implemented by the **magrittr** package. If you haven't loaded ggplot, dplyr, or wildlifeR yet you might have to load up magrittr directly.

```
library(magrittr)
```

For more background info on pipes see

- http://r4ds.had.co.nz/pipes.html
- https://seananderson.ca/2014/09/13/dplyr-intro/

When using **pipes** from **magrittr**, you start with data and follow it with an action you want done to it. So, for example, *previously* when we wanted the mean of the "mass"" column we did this

```
mean(my.frogs$mass) #[_]
```

Which is kind of read like a normal mathematical equation or function, where you start from inside the parentheses and work out.

Eg, this.is.read.2nd(this.is.read.1st)

R let's you nest as many functions as you want. If I want to round off my calculation I can wrap "mean(my.frogs$mass)" in "round(…)""

```
round(mean(my.frogs$mass)) #[_]
```

Using **pipes** to get the mean I write things more like a sentence

Eg, this.is.read.1st %>% this.is.read.2nd

```
my.frogs$mass %>% mean #[_] note parentheses after mean!
```

Which reads kind of like "Take the mass column and the dataframe and apply the **mean()** function to it."

To round the mean we would do this

```
my.frogs$mass %>% mean %>% round #[_]
```

Which read left to right like a sentence is "Take the mass column, calculate the mean, and then round off the mean"

Note that the round() command has an argument for how many digits you want to round to. You include that in the parentheses

```
my.frogs$mass %>% mean() %>% round(digits = 2) #[_]
```

16.5.1 Optional: Piping everything [O]

This section is optional

Most people learn about pipes when doing data summarizing and cleaning with dplyr and friends. But pipes can be used in many (most?) context.

Try this

```
my.frogs$mass %>% hist
```

Not everythign works though. For example, I can't figure out how to use pipes and t.test(). THere might be a way.

```
my.frogs %>% t.test(mass ~ sex)
```

End optional section

16.5.1.1 dplyr's summarize() command [__]

Instead of mean(data$column) we can use **summarise()** (for the British) or **summarize()**, plus pipes.

We can get the grand mean of just the mass column by loading **dplyr** using **library()** and then using the **summarise()** command

```
library(dplyr)                    #[_]
my.frogs %>% summarise(mean(mass))
```

This is maybe more complicated than "mean(my.frogs*mass*)*ormy*.*frogs*mass %>% mean, but overall the pipe framework and summarise pays off when combined with group_by() in the next section

16.6 dplyr's group_by() function [__]

For some more info on **group_by()** see

- https://www.r-bloggers.com/using-r-quickly-calculating-summary-statistics-with-dplyr/
- https://www3.nd.edu/~steve/computing_with_data/24_dplyr/dplyr.html http://www.datacarpentry.org/R-genomics/04-dplyr.html

We can use group_by() to split things up by a **categorical variable** (sex, color, year). Here, we can say "take my.frogs, split up the data by the sex column, and apply the mean() function to each subset."

```
my.frogs %>%            #[_] the data
  group_by(sex) %>%     #the group_by() function applied to the sex column
  summarise(mean(mass)) #the mean() function, applied to mass.
```

This might be a bit abstract when you first do it. Again, where starting with our whole dataframe (my.frogs), then its piped with %>% over to group_by() function, which splits it essentially into a male and a female sub-dataset. Then these two subsets are piped again to mean(). Then the mean() function is applied to the mass column in each of these subsets.

Note that the column heading in the output `mean(mass)`, which is what is in summarise().

Also note that the output is a "tibble", which is a common feature of the tidyverse. One thing that tibbles do is some reasonable rounding for you automatically.

A handy thing about summarise() is you can pass it labels. The following code adds a sensible label by changing "summarise(mean(mass))" to "summarise(mass.mean = mean(mass))", where "mass.mean = …" defines the label.

```
my.frogs %>%                        #[_]
  group_by(sex) %>%
  summarise(mass.mean = mean(mass))
```

You can label things anything, eg "puppies".

```
my.frogs %>%                      #[_]
  group_by(sex) %>%
  summarise(puppies = mean(mass))
```

When I first started using dplyr I found this syntax confusing because people often use the name of the function for the name of the column heading, like this

```
my.frogs %>%                       #[_]
  group_by(sex) %>%
  summarise(mean = mean(mass))
```

For some reason this trips me up because the word "mean" as a label here is not quoted; to me the word mean should be reserved for the function mean(). Also, in general you should always make both input and output output self-lableing. If you just look at the output, its not obvious what the "mean" being shown is.

You can pass any summary function to summarise(). We can give it **sd()** to get the sd of mass by sex. Note that I define the column names using "mass.sd = …"

```
my.frogs %>%                        #[_]
  group_by(sex) %>%
  summarise(mass.sd = sd(mass))
```

What makes dplyr::group_by and summarize() really powerful is that you can pass it *multiple*. summary functions at the same time. Here, I'll pass mean() and sd(), naming both.

```
my.frogs %>%                           #[_]
  group_by(sex) %>%
  summarise(mass.mean = mean(mass), #give me the mean!
            mass.sd = sd(mass))     #give me the sd!
```

dplyr also has a handy function **n()** for getting your sample size.

```
my.frogs %>%                          #[_]
  group_by(sex) %>%
  summarise(mass.mean = mean(mass),
            mass.sd = sd(mass),
            n = n())
```

16.6.1 OPTIONAL: Using novel functions with dplyr [O]

This section is optional

If you have defined the my_sd1() function above you can pass it to summarise() too.

```
my.frogs %>%
  group_by(sex) %>%
  summarise(mass.mean =  my_sd1(mass))
```

End optional section

16.7 OPTIONAL: Alternatives to dplyr

This section is optional

Most everybody is switching to dplyr. Below are some other idioms you may see others use or encounter in older books (or my old code).

16.7.1 doBy::summaryBy

The **doBy** package has a nice syntax. I don't really see many people use it. Be sure to download it first.

```
library(doBy)
summaryBy(mass ~ sex,data = my.frogs, FUN = mean)

summaryBy(mass ~ sex,data = my.frogs, FUN = c(mean,sd))
```

16.7.2 tapply()

tapply is pretty old school. You might see people working with big dat get into arguements about whether its faster than dplyr.

```
tapply(X = my.frogs$mass,INDEX = my.frogs$sex, FUN = mean)
```

16.7.3 reshape2::dcast

What I've used most of my career thus far. Am slowly switch to dplyr.

```
library(reshape2)
dcast(data = my.frogs,
      formula = sex ~ .,
      value.var = "mass",
      fun.aggregate  = mean)
```

End optional section

Chapter 17

Data analysis encounter: Graphical Data Exploration with Boxplots Using ggpubr

Nathan Brouwer, Phd
brouwern@gmail.com
https://github.com/brouwern
Twitter: lobrowR

17.1 Introduction

- In the 1st few sections we explored several graphing approachs (plot, ggpubr, ggplot::qplot, ggplot)
- Now we will focus on the ggpubr extension of ggplot
- We'll use ggpubr::ggboxplot to graphically explore our data

17.1.1 Goals & objectives

Create plots to explore variation in the frogarms data, with an emphasis on boxplots using ggpubr::ggboxplot().

17.1.2 Packages

- ggplot2
- cowplot
- ggpubr
- dplyr

17.1.3 Outline

- load packages if necessary
- load data and subset data if necessary
- make data exploration graphs
- boxplots, hinged boxplots, boxplots with raw data

17.1.4 Vocab

- jittering
- boxplot
- R object
- categorical variable
- continuous variable
- faceting

17.1.5 Preliminaries

17.1.5.1 Load packages

If not already loaded, we need the **wildlifeR** package, which lives on Github

```
library(devtools)
install_github("brouwern/wildlifeR")

library(wildlifeR)
```

We also need several other packages for visualization. We'll use the **ggplot2** package for plotting, and the **cowplot** package for some nice plotting defaults. **cowplot** also has a hand function for putting two plots into the same graph.

```
library(ggplot2)
library(cowplot)
library(ggpubr)
library(dplyr)
```

17.1.5.2 Load data

Load the frogarms data by Buzatto et al (2015) if its not already loaded.

```
data(frogarms)
```

17.1.5.3 Subset your data

The function make_my_data2L() will extract out a random subset of the data. Change "my.code" to your school email address, minus the "@ pitt.edu" or whatever your affiliation is. **This does not need to be done if you did this as part of the previous lesson**

```
my.frogs <- make_my_data2L(dat = frogarms,
                           my.code = "nlb24", # <=  change this!
                           cat.var = "sex",
                           n.sample = 20,
                           with.rep = FALSE)
```

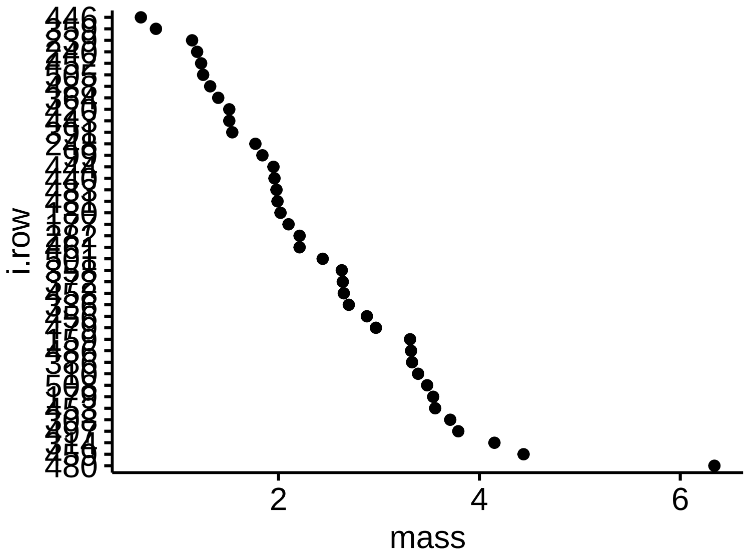
17.2 Data exploration plots

We'll make some plots to look at the overall structure and distribution of the data. After first looking at a **Cleveland dotplot**, we'll focus on **boxplots** but also consider some others, including **violin** plots.

17.2.1 Cleveland dot plot

A useful tool for looking at your data is a **Clevand dot plot.** This is just the variable you are interested in (eg mass) plotted against its rank in the data set or some other organization scheme. This allows you to quickly spot an values that are really weird, such as typos.

```
ggdotchart(data = my.frogs,
           x= "i.row",
           y = "mass",
           rotate = T)
```



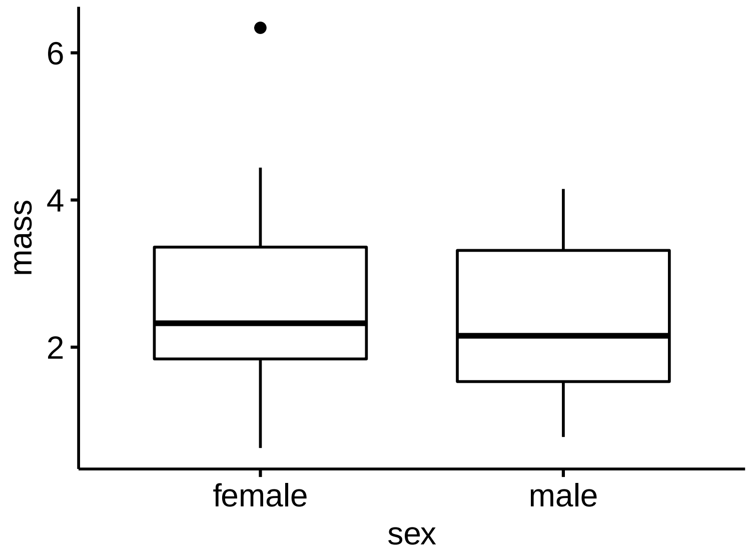
For more on Cleveland dotplots and data exploraiton see

Zuur et al 2010. A protocol for data exploration to avoid common statistical problems. Methods in E&E. https://besjournals.onlinelibrary.wiley.com/doi/full/10.1111/j.2041-210X.2009.00001.x

17.2.2 Boxplots

Basic boxplots are easy to make with ggpubr's **ggboxplot()** function. **Note that "mass" and "sex" are in quotes.**

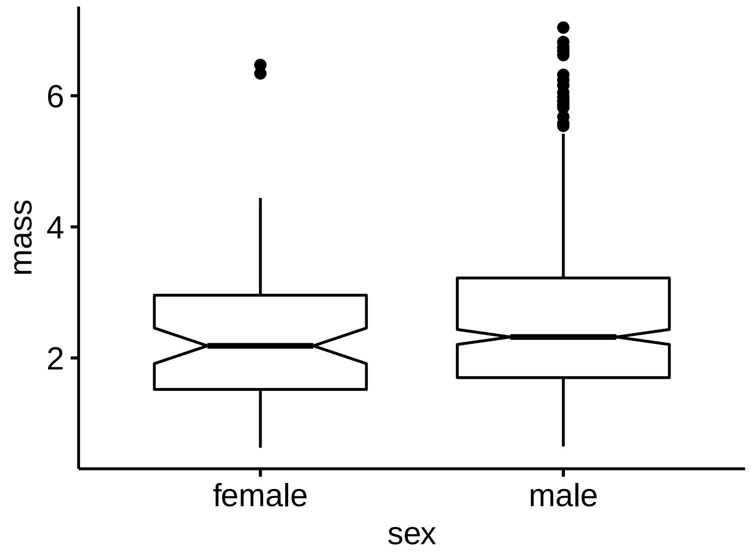
```
ggboxplot(data = my.frogs, # the data frame
          y = "mass",      # y-axis: a continous variable; in quotes!
          x = "sex")       # x-axis: a group; in quotes!
```



17.2.3 Notched boxplot

We'll use the original frogarms dataframe first for this. These aren't commonly used; the notches work kind of like confidence intervals to determine if medians are different.

```
ggboxplot(data = frogarms, #full dataset
          y = "mass",
          x = "sex",
          notch  = TRUE)
```



Now try your own subset of the data. The Notch calculations likely get messed up with small samples sizes. R will likely give you several warnings in red.

```
ggboxplot(data = my.frogs, #my subset
          y = "mass",
          x = "sex",
          notch  = TRUE)
```

17.2.4 Filled boxplots

Its good practice to accent plot elements that relate to different groups. **ggplot** and **ggpubr** make this really eas. We can add colored fill to the box plots using fill = "…"; note that it is "**fill**" not "color". (Color changes the color of the lines).

```
ggboxplot(data = my.frogs,
          y = "mass",      # quotes!
          x = "sex",
          notch  = TRUE,
          fill = "sex")
```

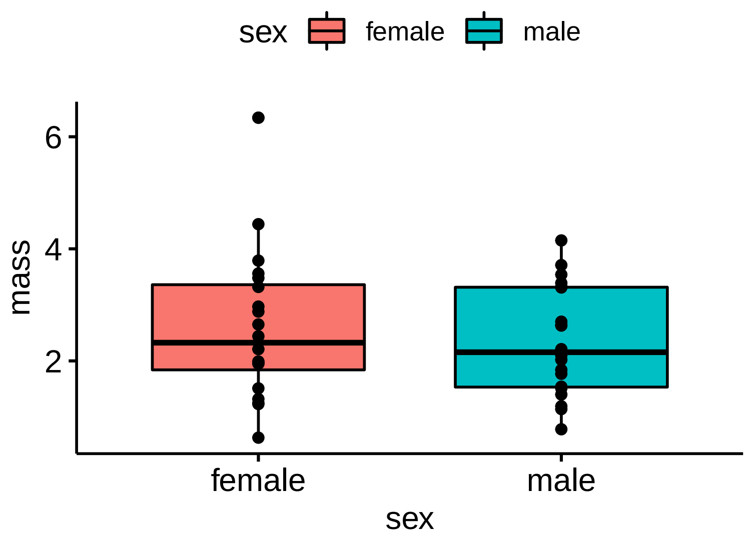
We can turn off the notching by adding a "#" character before it. This is called **commenting out** that line of code

```
ggboxplot(data = my.frogs,
          y = "mass",
          x = "sex",
          #notch  = TRUE,
          fill = "sex")
```

17.2.5 Boxplots with raw data

Its best to plot your raw data whenever possible. This works best with small datasets. We'll do this by appending add = "point".

```
ggboxplot(data = my.frogs,
          y = "mass",
          x = "sex",
          #notch  = TRUE,
          fill = "sex",
          add = "point")    # add = "point", with quotes!
```



For more best practices in plotting data see

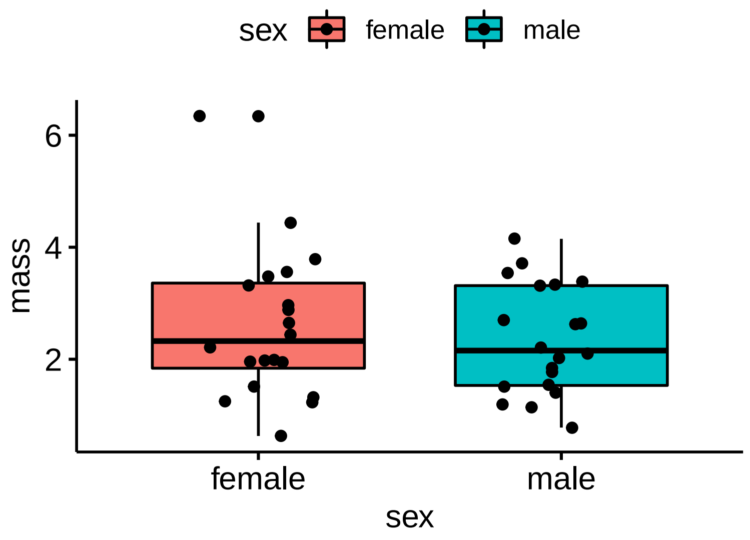
- Weissgerber et al. 2017. Data visualization, bar naked: A free tool for creating interactive graphics. Journal of Biological Chemistry. http://www.jbc.org/content/292/50/20592.full

- Weissgerber et al. 2015. Beyond bar and line graphs: time for a new data presentation paradigm. PLoS. https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1002128&fullSite

17.2.6 Boxplots with jittered raw data

This can be helpful, though ggpubr::ggboxplot doesn't allow much control over the **jittering**. Jittering is helpful when you have large datasets and want to avoid overlap in the points. We'll append add = "jitter".

```
ggboxplot(data = my.frogs,
          y = "mass",
          x = "sex",
          fill = "sex",
          add = "jitter") # add = "jitter"
```



17.2.7 OPTIONAL: Jittering with ggplot2 [O]

The following section is opptional

ggpubr helps simplify ggplot2 code, but in doing so adds some constraints. You can combine ggpubr commands with regular ggplot2 code though. We'll use the code we did above and also add "+ geom_jitter()"

This code should produce a plot similar to the one above. Note that after " fill = "sex") " there is a "+", (eg, " fill = "sex") + ") and that on the next line is " geom_jitter()"

```
ggboxplot(data = my.frogs,
          y = "mass",
          x = "sex",
          fill = "sex") +  # need the plus!
  geom_jitter()            # the jitter command
```

We can make the jittering less extreme by adding "width = 0.1" within geom_jitter()

```
ggboxplot(data = my.frogs,
          y = "mass",
          x = "sex",
          fill = "sex") +  #need the plus!
  geom_jitter(width = 0.1) #reduce the magnitude of the jitter
```

End optional section

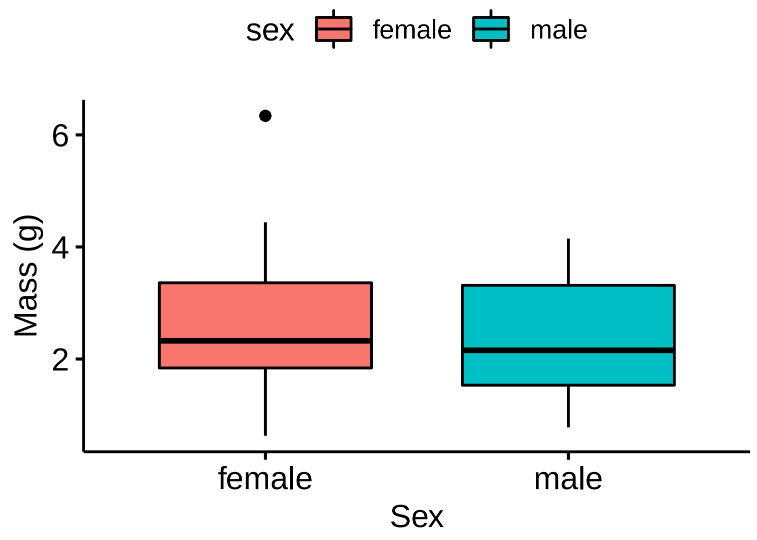
17.2.8 Label ggpubr axes

A graph isn't done until it has labels. This can get annoying in base R (eg plot()) graphics and ggplot2, but is easy in ggpubr.

17.2.8.1 Adding Axes lables

Adding the arguements "xlab = …" and "ylab = …" adds axes labels. Always add units (eg "g" for grams) when applicable.

```
ggboxplot(data = my.frogs,
          y = "mass",
          x = "sex",
          fill = "sex",
          xlab = "Sex",       #x axis (horizontal); quotes!
          ylab = "Mass (g)") #y axis (vertical)
```



17.2.8.2 Plot title

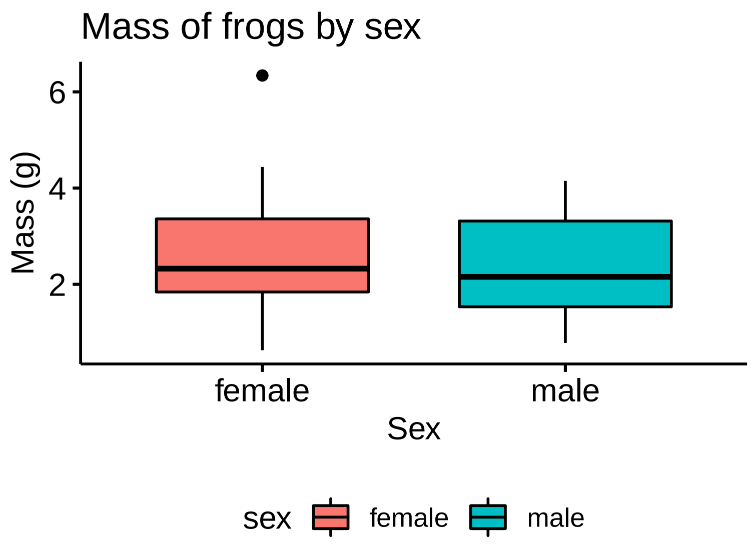
The command "main = …" adds a main title at the top of the graph. This is not usually done for publications but useful for keeping track of things and for presentations.

```
ggboxplot(data = my.frogs,
          y = "mass",
          x = "sex",
          fill = "sex",
          add = "jitter",
          xlab = "Sex",
          ylab = "Mass (g)",
          main = "Mass of Australian frogs by sex") #Main title
```

17.2.8.3 Refining ggpubr plots

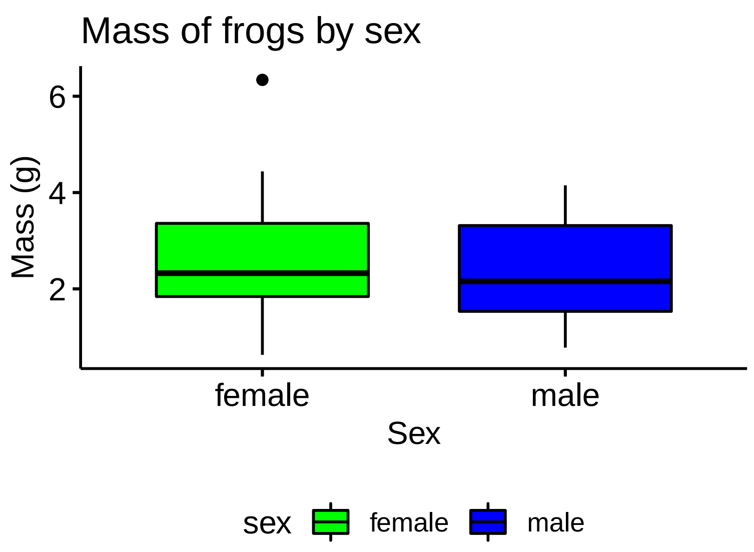
Move the legend to the bottom using **legend = "bottom"**.

```
ggboxplot(data = my.frogs,
          y = "mass",
          x = "sex",
          fill = "sex",
          xlab = "Sex",
          ylab = "Mass (g)",
          main = "Mass of frogs by sex", # main title
          legend = "bottom")             # location of legend
```



Change the color palette; Note that the colors are within **c(...)**.

```
ggboxplot(data = my.frogs,
          y = "mass",
          x = "sex",
          fill = "sex",
          xlab = "Sex",
          ylab = "Mass (g)",
          main = "Mass of frogs by sex",
          legend = "bottom",
          palette = c("green","blue"))  # change pallete
```



17.2.9 Plotting multple plots with cowplot::plot_grid

You can save just about anything as an R object, and we can save a plot to an **R object**. I will use the **assignment operation** (<-) to assign the output of ggboxplot() to an object called "gg.my.frogs". Note that here I am using my.frogs.

```
gg.my.frogs <- ggboxplot(data = my.frogs,
          y = "mass",
          x = "sex")
```

Note that the code runs but nothing happens...

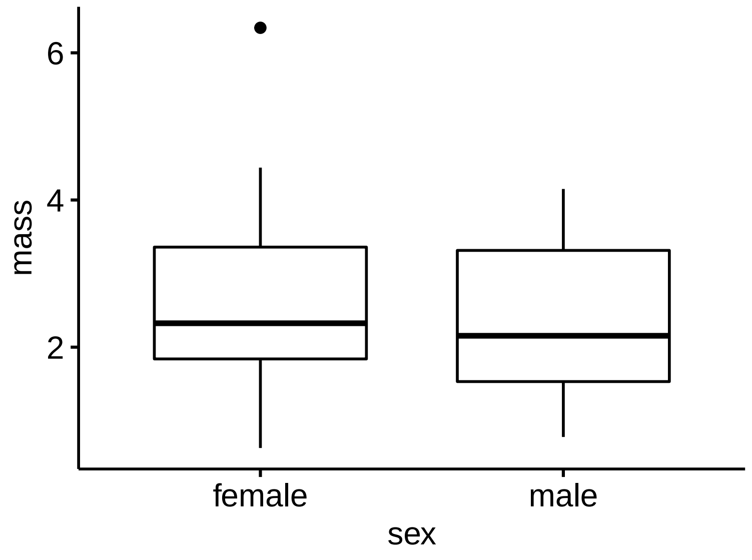
We can see what we just made using is()

```
is(gg.my.frogs)
```

This indicates 1) gg.my.frogs is there and 2) it is a "gg" type R object.

I can get the graph if I call just the object gg.my.frogs (eg, just type "gg.my.frogs" into the console and press enter, or highlight just the word "gg.my.frogs" in a script and execute the command)

```
gg.my.frogs
```

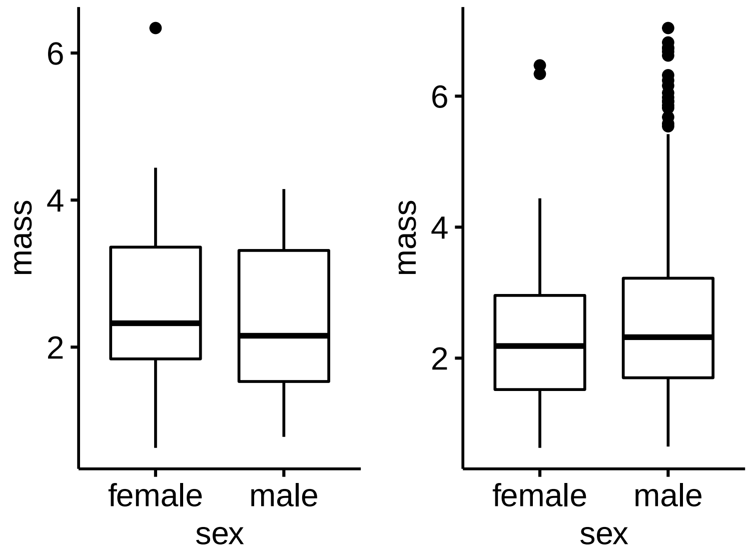


Now, Make an object using the full frogarms data

```
gg.frogarms <- ggboxplot(data = frogarms, #use original data
          y = "mass",
          x = "sex")
```

Now plot both using the plot_grid() function from the handy cowplot package.

```
plot_grid(gg.my.frogs,
          gg.frogarms)
```



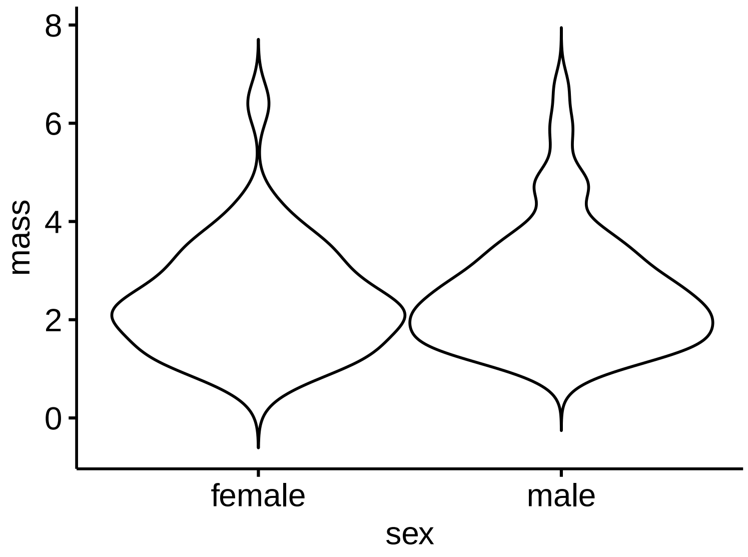
Add labels. Note that alignment is off sometimes.

```
plot_grid(gg.my.frogs,
          gg.frogarms,
          labels = c("a)My fogs","b)All the frogs"))
```

17.2.10 Violin plots

An interesting alternative to boxplot that works well with large data is the violin plot

```
ggviolin(data = frogarms, #use original data
         y = "mass",
         x = "sex")
```



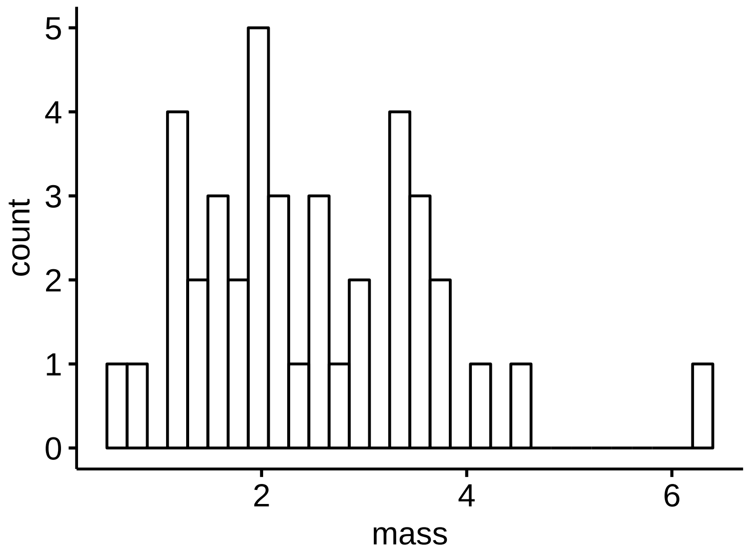
17.2.11 Optional: Histograms [0]

The following is optional

Histograms are excellent for data exploration. They generally work best with medium to large datasets.

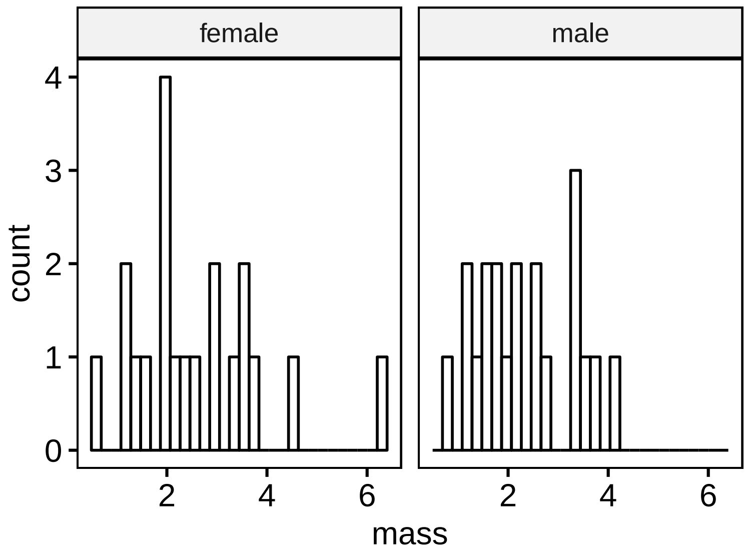
A basic histogram can be made using gghistogram(). Note that there is "x = …" but no "y = …"; the y-axis is computed by the graphing function.

```
gghistogram(data = my.frogs,
            x = "mass")
```

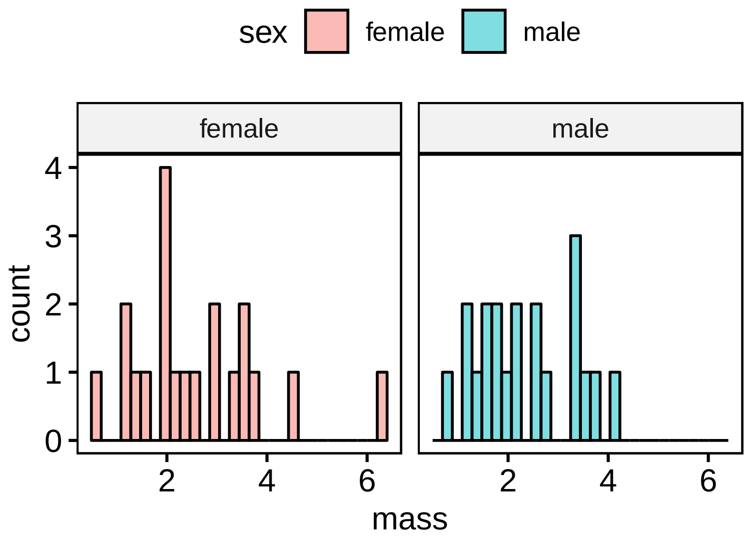


A key concept for **ggplot** is **faceting**. Faceting occurring when a two panels of plots are made from a single dataset, and the panels are split by a **categorical variable**. We can add the arguement **"facet.by.by = sex"** to make 2 panels, one for female and one for male. Note that because there are only 10 frogs in each group, the graphs aren't very useful.

```
gghistogram(data = my.frogs,
            x = "mass",
            facet.by = "sex")
```



Just as we did for histograms we can change the fill, add a title, etc.



17.3 Challenge: Compare a boxplot and violin plot [_]

Using ggboxplot, ggviolin, and plot_grid, make a 2-panel plot that compare a boxplot to a violin plot. Add the raw jittered raw data to both plots.

Chapter 18

Data analysis encounter: Plotting means and measures of variation and precision

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter: lobrowR

18.1 Introduction

In the previous lesson we made **boxplots** to explore the distribution of the data. In this lessons we'll focus on plotting the mean with error bars. Different kinds of error bars are possible, including the **standard deviation**, **standard error**, and **confidence intervals**. Norms for what to plot vary between fields; generally speaking standard error are most common in biology, but 95% CIs are what are recommended by statisticians. As we'll explain, standard deviations convey one kind of information, while standard errors and confidence intervals convey a very different kind.

18.1.1 Goals & objectives

Create plots to explore the **wildlifeR::frogarms** data and visualize both **variation within** the groups using the standard deviation, and **how precisely the mean can be estimated** using standard errors (SE) and confidence intervals.

18.1.2 Packages

- ggplot2
- cowplot
- ggpubr
- dplyr

18.1.3 Outline

- Preliminaries (if needed): load packages, load data, subset data
- Represent variation with the standard deviation
- Represent precision of estimated means using SE and 95% CIs

18.1.4 Vocab

- standard deviation
- standard error
- confidence intervals

18.1.5 Preliminaries

Note: The following code doesn't need to be run if the previous exercises in this section were already run.

18.1.5.1 Load packages

If not already loaded, we need the wildlifeR package, which lives on Github

```
library(devtools)                    #[P]
install_github("brouwern/wildlifeR")

library(wildlifeR)
```

We also need several other packages needed for visualization. We'll use the ggplot2 package for plotting, and the cowplot package for some nice plotting defaults.

```
library(ggplot2) #[P]
library(cowplot)
library(ggpubr)
library(dplyr)
```

18.1.5.2 Load data

Load the frogarms data by Buzatto et al (2015) if its not already loaded.

```
data(frogarms) #[P]
```

18.1.5.3 Subset your data

The function make_my_data2L() will extact out a random subset of the data. Change "my.code" to your school email address, minus the "?" or whatever your affiliation is. **This does not need to be done if you did this as part of the previous lesson**

```
my.frogs <- make_my_data2L(dat = frogarms,     #[P]
                           my.code = "nlb24", # <=  change this!
                           cat.var = "sex",
                           n.sample = 20,
                           with.rep = FALSE)
```

18.2 Background: measures of variation [__]

18.2.1 Variance

The **variance** is a ubiqitous metric which quantifies the amount of variability in a given set of data. The equation in all its glory is:

$$s^2 = \frac{\sum_{i=1}^{n} \left(Y_i - \bar{Y}\right)^2}{n-1}$$

This can be disorienting if you aren't used to looking at this notation. We'll go over it for the sake of being thorough; we mostly need a general conceptual understanding of the variance right now.

Y_i represents each data point, where "i" stands for "index". Y_1 is the 1st value in the dataset, Y_2 is the second values in the data set etc. The Y with the bar over it, $\bar{Y}$, is called "**Y bar**" and can be typed out as "Y.bar" in R. **n** is the sample size. $\sum$ is the summation operator (the Greek letter sigma). $\left(Y_i - \bar{Y}\right)$ is the difference between each value in the data (the Y.i) minus the mean value (the Y.bar).

$(Y_i - \bar{Y})^2$ indicates that each difference between a value and the mean should be squared. These are called the squared differences or squared deviations. $\sum(Y_i - \bar{Y})^2$ indicates that all of these values should be summed together; this is called "**the sum of squares**". Finally, the sum of is divided by n-1.

So, the upshot is: we take each value in the dataset, subtract the mean from each value, square each of those differences, add them up, and divide them by the sample size minus 1.

The variance shows up *everywhere* in stats, but mostly behind the scenes in calculations; less frequently is it reported, though it plays an important role in, among other things, genetics and stochastic demography.

The bigger the variance, the more variability. However, because of the squaring that occurs in the numerator the variance is on scale all its own. The variance is therefore never plotted alongside the datae; this would be meaningless.

The variance of the frogarm mass data is

```
var(frogarms$mass)
```

18.2.2 Standard deviation (SD)

The variance is a bit hard to wrap you mind around; easier and much more frequently reported in practice is the **standard deviation**. The standard deviation is just the square root of the variance.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} \left(Y_i - \bar{Y}\right)^2}{n-1}}$$

Taking the square root of the variance puts it back into the same scale as the raw data. The standard deviation can therefore be plotted alongside the raw data or the mean.

THe standard deviation for the frogarm mass data is

or equivalently

```
sd(frogarms$mass)
```

Since the standard deviation is the square root of the variance, it is always smaller than the variance (Eg s < s^2)

The standard error (and the variance) are both measures of **variation**. They indicate the amount of variation that occurs in the **raw data.**

18.2.3 Optional: Testing the numerical equivalence of 2 values

The following is optional

We can confirm that 2 values are numerically equivalent using the == operator:

```r
sqrt(var(frogarms$mass)) == sd(frogarms$mass)
```

End optional section

18.2.4 Standard error (SE)

The **standard error** is often neglected in elementary stats work but it is a central concept. The standard error is the standard deviation divided by the square root of the same size.

$$SE = \frac{SD}{\sqrt{N}}$$

A very important thing about the standard error: it is **not** a measure of variation. It does **not** indicate how much variation there is in the raw data. The SE is an indication of **precision**. Specifically, it indicates how confident we are about our estimate of mean from the data. The SE does depend on the amount of variation in the data, but also depends on the sample size (n).

Base R does not have a function for the SE. We can calculate it like this

```r
sd(frogarms$mass)/sqrt(length(frogarms$mass))
```

This is a bit dense, so we can break it up. First, assign the numerator and denominator to seperate **R objects** using the assignment operator **<-**.

```r
frog.sd <- sd(frogarms$mass)
frog.n  <- sqrt(length(frogarms$mass))
```

Then do the division

The standard error is frequently used as **error bars** for plots of means.

18.2.5 Confidence Interval (CI)

Confidence intervals are a deep topic in stats. We'll just touch on them briefly. If you have calcualted the standard error (SE) you can *approximate* the 95% CI for a mean as

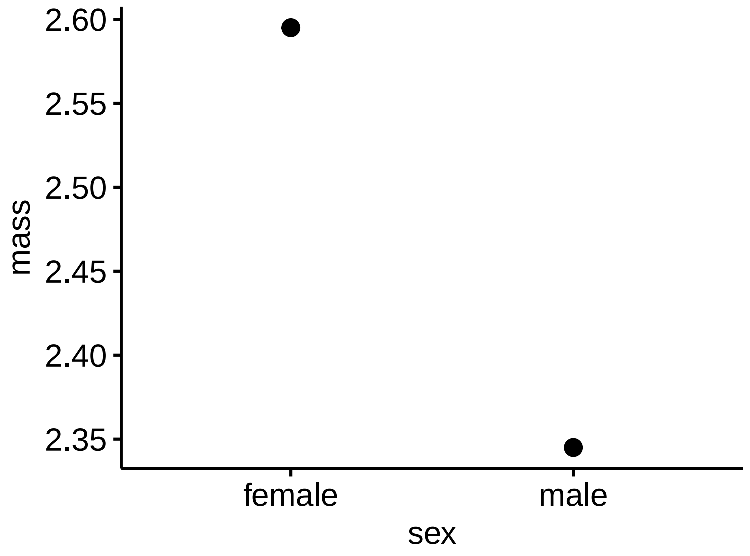
$$CI = 1.96 * SE$$

That is, 1.96 (sometimes rounded just to 2.0) multipled by the SE. The 95% CI is argueably a better choice for an error bar around a mean than the SE (better yet plot both, which down the road we'll show how to do).

18.3 Representing variation with the SD

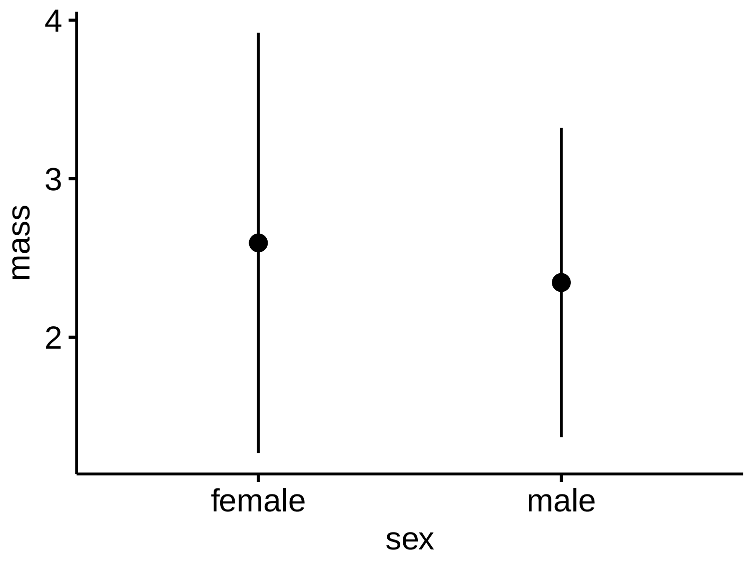
ggpubr has a very hand function for calcualting means and plotting error bars around them. **ggerrorplot()** (gg error plot) is the main function, and the "**desc_stat = ...**" arguement defines what exactly to plot.

If for some reason you just want to plot means use "**desc_stat = 'mean'**", with mean quoted.



For the standard deviation use "**desc_stat = 'mean_sd'**" (Note that any time you have a plot with errorbarrs – and if you plot means you should have error bars – you need to define at least in the figure legend what the error bars are.)

```
ggerrorplot(data = my.frogs,
            desc_stat = "mean_sd",
          y = "mass",
          x = "sex")
```

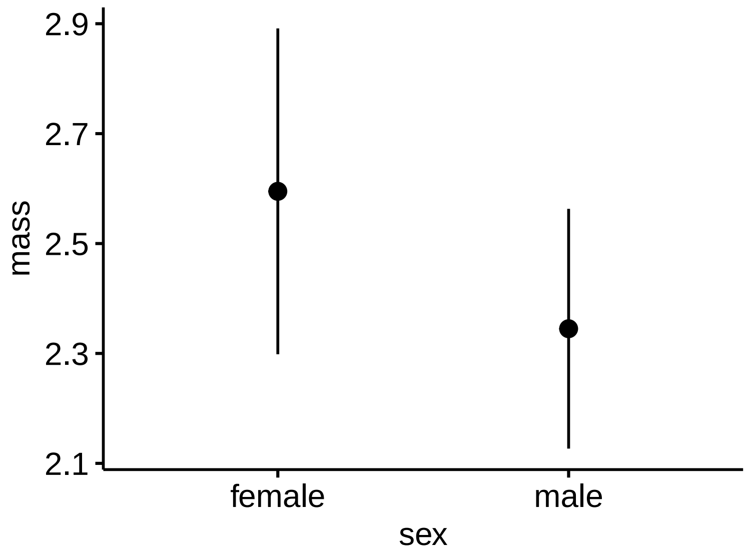


Again, the standard deviation is a measure of **variation**.

18.3.1 Representing precision with the SE

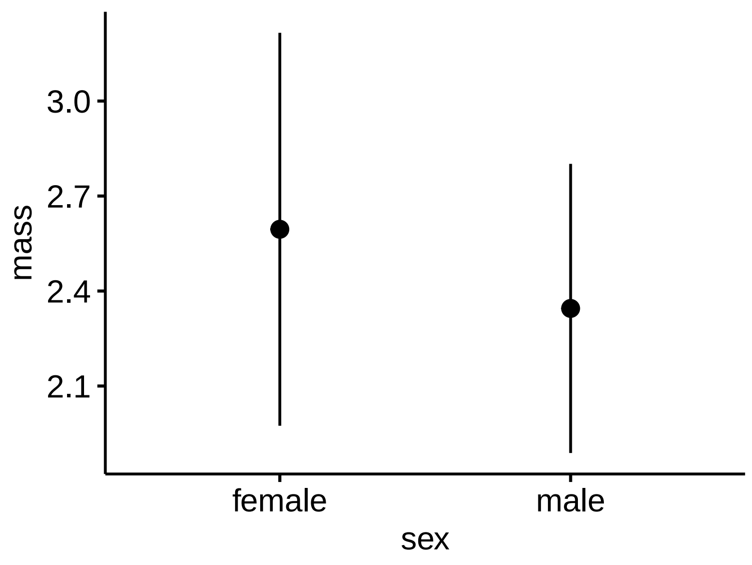
ggerrorplot() actually defaults to making a plot of the mean +/- 1 standard error (SE).

```
ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex")
```



The means and 95% confidence interval are plotted with “**desc_stat = ‘mean_ci’**”

```
ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex",
          desc_stat = "mean_ci")
```



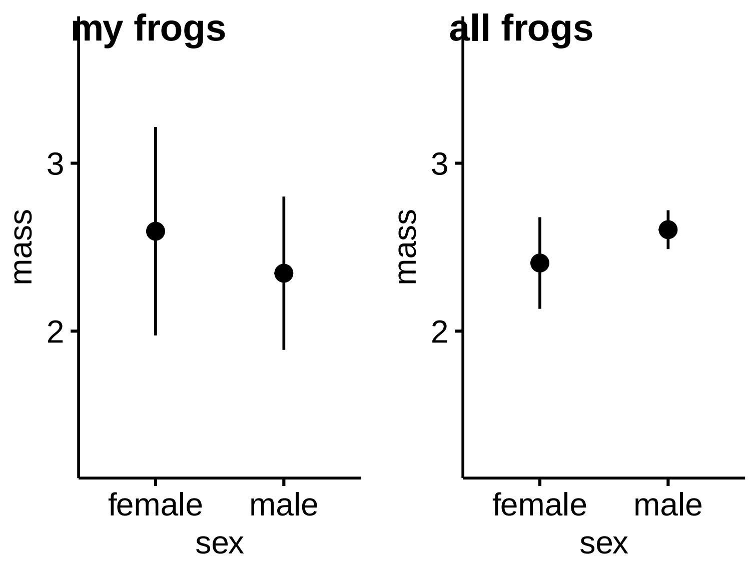
Again, the SE and 95% CI are measure of **precision**. There can be lots of variation in a dataset (SD is high) but if you have collected a lot of data (N is arge), you should be able to estimate the mean with precision. In general, the more data you have, the more precise your estimate will be.

You can see how more data increases precision by comparing the entire frogarms dataset against your personal subset. We can assign each plot to an object using the assignment operator "<-" and then plot them side by side with **plot_grid()** from **cowplot**.

```r
#your data
gg.my.frogs <- ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex",
          desc_stat = "mean_ci",
          ylim = c(1.25,3.75))

#all of the forgm data
gg.all.frogs <- ggerrorplot(data = frogarms, #changed data = ...
          y = "mass",
          x = "sex",
          desc_stat = "mean_ci",
          ylim = c(1.25,3.75))
```

Now plot them together using **cowplot::plot_grid()**. Then means will be different because of random variation between the subsamples. What happens to the error bars?.

```r
plot_grid(gg.my.frogs,
          gg.all.frogs,
          labels = c("my frogs","all frogs"))
```

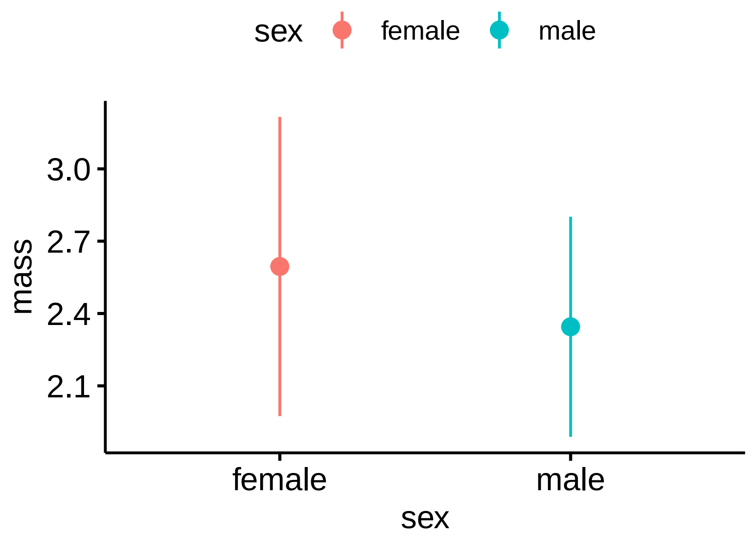


Now, what if instead of the 95% CI we plotted the SD? What do you think that will look like? Adapt the code from above by changing "**desc_stat = 'mean_ci'**" to "**desc_stat = 'mean_sd'**".

18.4 Refining plots

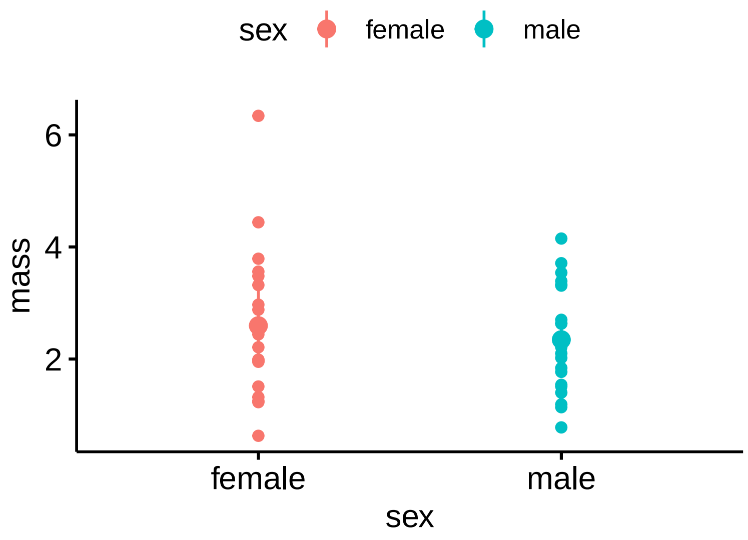
Set colors

```
ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex",
          desc_stat = "mean_ci",
          color = "sex")         # color = ....
```



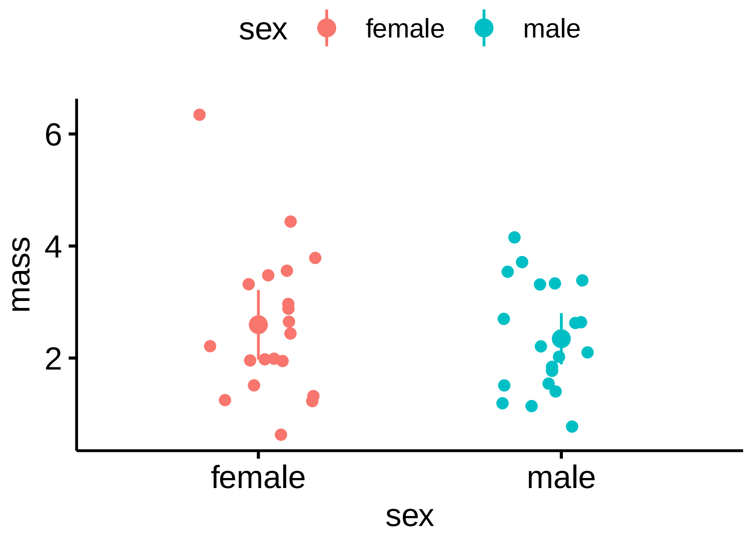
Add raw data. kinda crazy but worth seeing how it looks.

```
ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex",
          desc_stat = "mean_ci",
          color = "sex",
          shape = "sex",
          add = "point")  # add = "point"
```



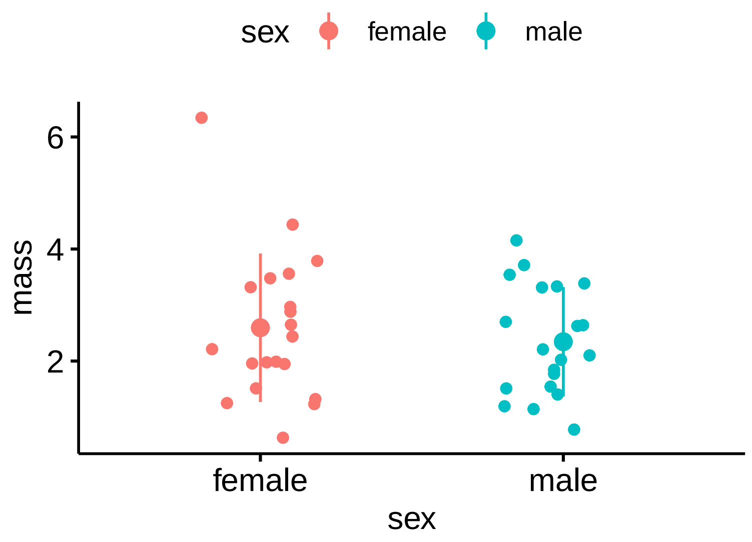
Jitter raw data. Even crazier. In general it works best if there are less than 10 data points per group.

```
ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex",
          desc_stat = "mean_ci",
          color = "sex",
          add = "jitter")        # add = "jitter"
```



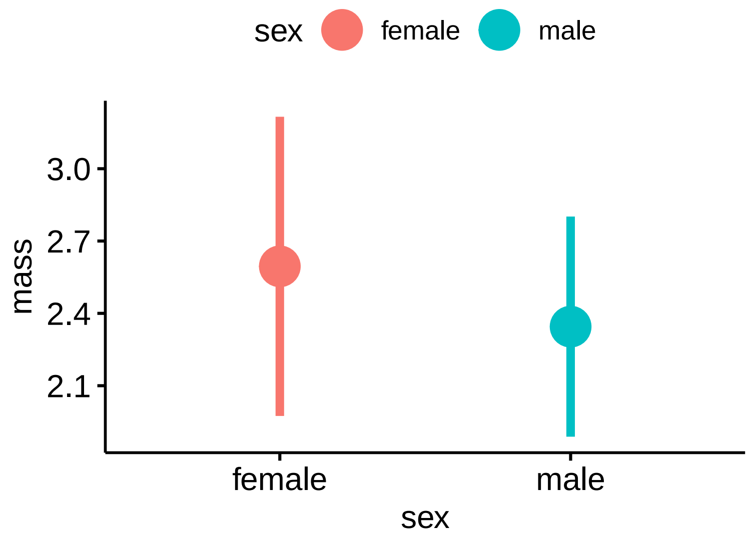
Change to the SD. In theory, about 2/3 of the data points should fall within +/- 1 SD. Does that look about right?

```r
ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex",
          desc_stat = "mean_sd",   # desc_stat = "mean_sd"
          color = "sex",
          add = "jitter")
```



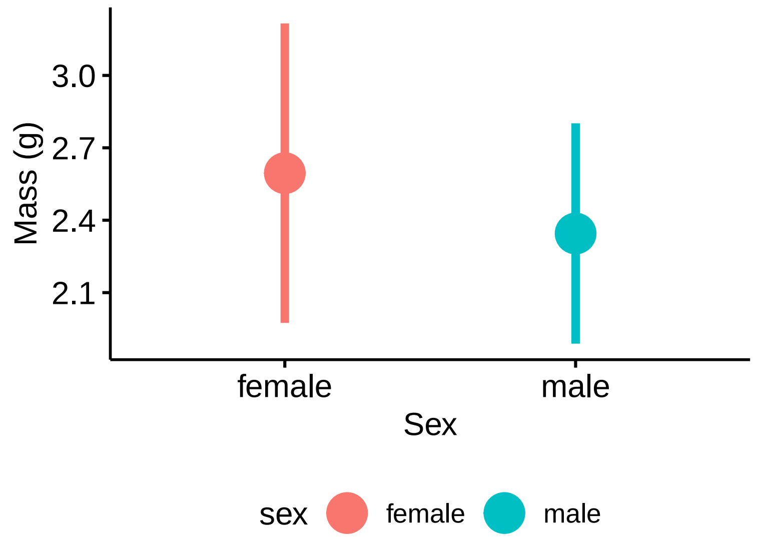
Back to the means and CIs, and increase size of the points.

```r
ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex",
          desc_stat = "mean_ci",
          color = "sex",
          size = 1.5)            #increase point size
```



Move legend to the bottom using “legend =”bottom” “. Add some labels using xlab and ylab.

```r
ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex",
          desc_stat = "mean_ci",
          color = "sex",
          size = 1.5,
          xlab = "Sex",
          ylab = "Mass (g)",
          legend = "bottom")
```



Chapter 19

Data analysis encounter: T-test

Nathan Brouwer, Phd
brouwern at gmail.com
https://github.com/brouwern
Twitter lobrowR

19.0.1 Preliminaries

19.0.1.1 Load packages

```r
library(wildlifeR)
library(ggplot2)
library(cowplot)
library(ggpubr)
library(dplyr)
```

19.0.1.2 Load data

```r
data(frogarms)
```

19.0.1.3 Subset your data

The function make_my_data2L() will extract out a random subset of the data. Change "my.code" to your school email address, minus the "?" or whatever your affiliation is.

```r
my.frogs <- make_my_data2L(dat = frogarms,
                           my.code = "nlb24", # <=  change this!
                           cat.var = "sex",
                           n.sample = 20,
                           with.rep = FALSE)
```

t-test used to tell if 2 groups are different

19.0.2 T-test

Use the t.test() function. The **reponse variable** (aka the **y variable**) is **mass** and goes to the left of the ~. The **predictor** variable (**x-variable**) goes on teh right. The arguement **data = ...** is used to tell the function where the response and predictor columns are found.

This spits out R's standard t.test table

Save to object

```
mass.t <-  t.test(mass ~ sex, data = my.frogs)
```

19.0.3 Optional: cleaning up output with broom() [O]

This section is optional

look at w/broom::glance. re-labels things a bit odd and would be nice to round. will stick with original R output

```
library(broom)
glance(mass.t)
```

End optional section

```
mass.t
```

Check the means using dplyr

```
my.frogs %>% group_by(sex) %>% summarize(mean.mass = mean(mass))
```

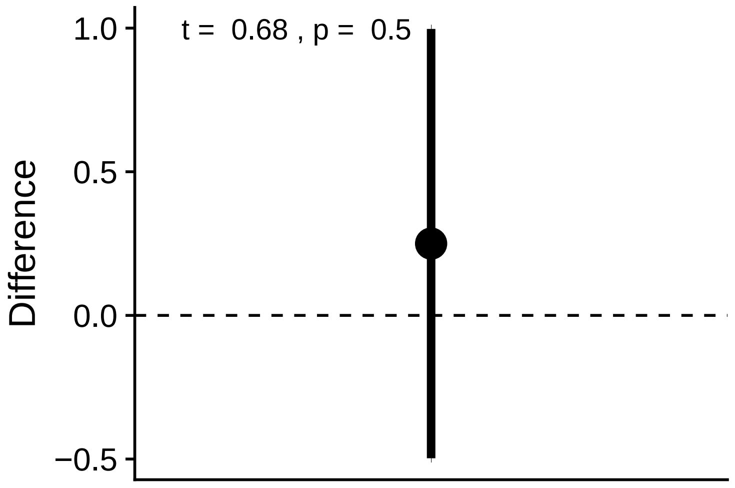
What does all of this mean?

Quiz

p = p interpretation = df = why df fractional? (this one is hard!) t = What would happen to p if t was bigger? What is a "95% CI" What is this a 95% CI for?

What does the CI mean?

```
plot_t_test_ES(mass.t)
```



Make a plot of the means with error bars. Save to an object called gg.means

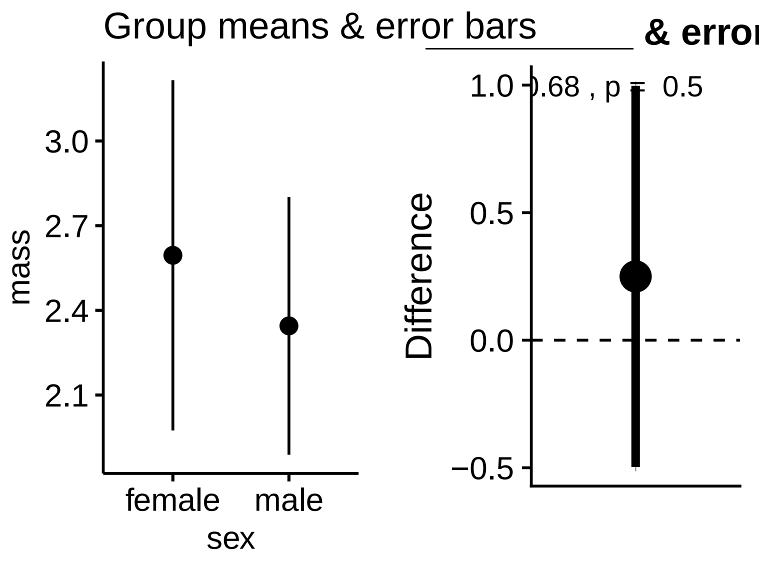
```
gg.means <-ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex",
          desc_stat = "mean_ci") +
  ggtitle("Group means & error bars")
```

Save effect size

```
gg.ES <-plot_t_test_ES(mass.t) +
  ggtitle("__________ & errorbars")
```

Plot both

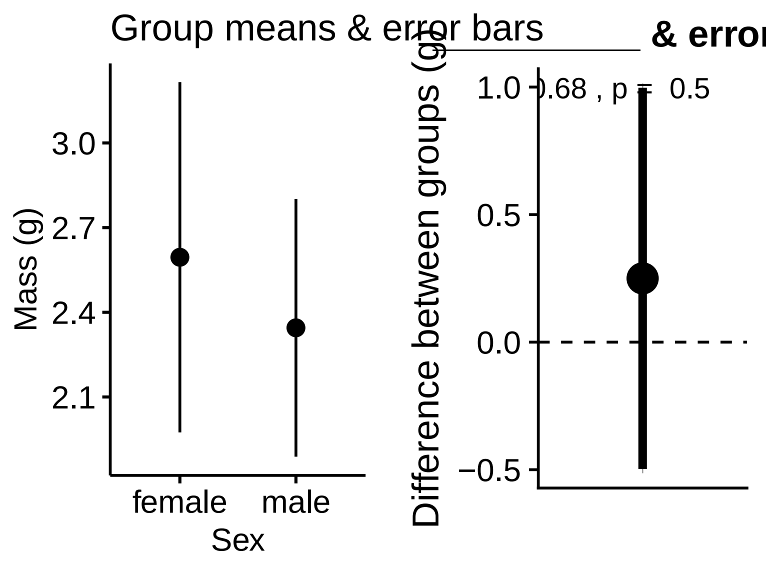
```
plot_grid(gg.means,gg.ES)
```



A hint

```
gg.means <-ggerrorplot(data = my.frogs,
          y = "mass",
          x = "sex",
          desc_stat = "mean_ci") +
  ylab("Mass (g)") +
  xlab("Sex") +
  ggtitle("Group means & error bars")


gg.ES <-plot_t_test_ES(mass.t) +
  ylab("Difference between groups (g)") +
  ggtitle("__________ & errorbars")
```

```
plot_grid(gg.means,gg.ES)
```



Did anyone get a significant result?

19.0.4 Arm girth

(do same thing for arm girth. see the super significant values?)

Now we are going to unpack this

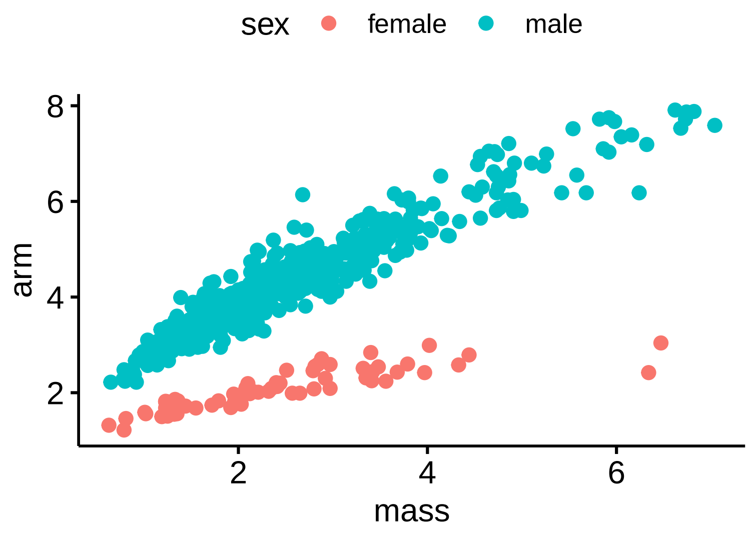
19.0.5 Optional

This section is optional

There is a **major** flaw in this analysis. consider the following graph where the mass is plotted on the x-axis and the arm girth is plotted on the y-axis. Does arm girth vary just because of sex, or because of sex and mass?

This is ANCOVA. Sometimes taught as an extension of ANOVA, or as a type of regression.

```
ggscatter(data = frogarms,
          y = "arm",
          x = "mass",
          color = "sex")
```



End optional section

Bibliography