# Introduction to R

Data & Software Carpentry

The term `R` is used to refer to both the programming language as well as the software that interprets the scripts written using it. The learning curve may be steeper than with other statistical software, but with `R` the results of your analysis or your plot does not rely on remembering what order you clicked on things, but instead on the written commands you generated. In `R` you will work in scripts or with dynamic documents, with scripts within them (Rmd or Rnw files). Scripts may feel strange at first, but they make the steps you used in your analysis clear for both you and for someone who wants to give you feedback, further promoting the importance of reproducible science!

RStudio is a free computer application that allows you access to the resources of `R`, while providing you with a comfortable working environment. There are many ways you can interact with `R`, but for many reasons RStudio has become the most popular. To function correctly, RStudio uses `R` behind the scenes, hence both need to be installed on your computer. Both `R` and **`RStudio`** are cross-platform, so that everyone's versions look and operate the same regardless of their operating system!

For our workshop, we will be making use of the RStudio Cloud. The RStudio Cloud functions and looks the same as the RStudio you can install on your computer, but comes with some perks for collaboration and easily accessing the workshop materials. If you so wish, you can choose to download a local copy of RStudio on your personal computer. There are many videos and guides for installation. See the Stat 217 textbook (pages 12-14) here: https://scholarworks.montana.edu/xmlui/bitstream/handle/1/2999/Greenwood_Book. pdf?sequence=3&isAllowed=y_
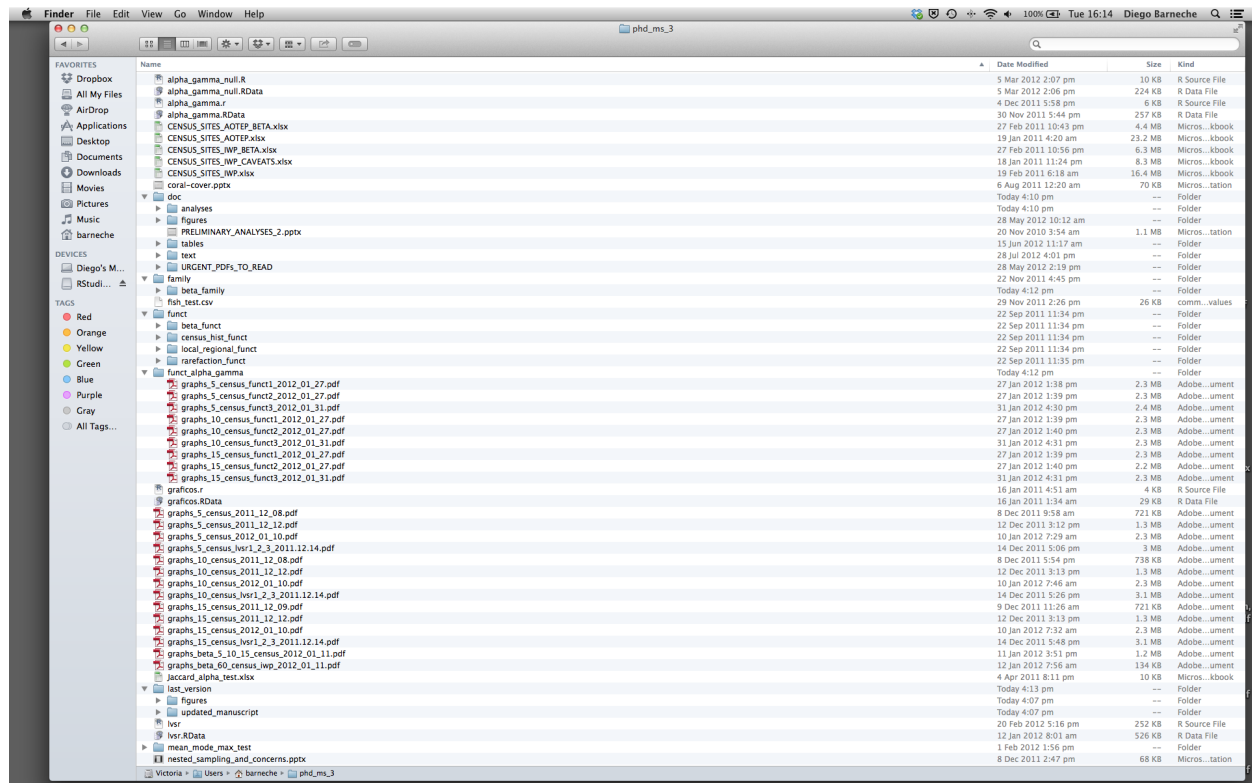
RStudio has a panel of 4 windows, where each can be viewed at the same time and has multiple tabs available.

- the **Editor** for your scripts and documents (top-left)

- the `R` **Console** (bottom-left)

- your **Environment (Objects/Variables)/History** (top-right)

- and your **Files/Plots/Packages/Help/Viewer** (bottom-right).
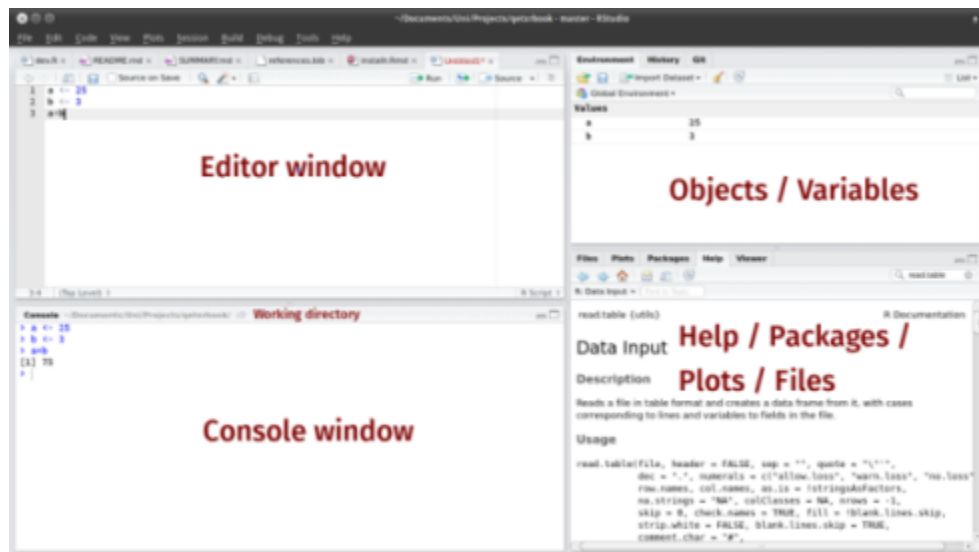
Work Flow in R

Many people tend to organize their projects like this:

There are many reasons we should *always* avoid this shamble of organization:

- it is really hard to tell which version of your data is original and what versions are modified

- things get really messy because all types of files are mixed together

- it's probably hard for you to find things and relate the correct files to their respective output

Ultimately, good project organization will make your life easier! It helps ensure the integrity of your data, makes it simpler to share your code or get help with your code, allows for you know exactly what code you used on a paper, and it's easier to pick a project back up.

It is good code writing and file storage practice to keep a set of all related data, analysis, plots, documents, etc. in the same folder. RStudio makes this process easy with the invention of projects. In an RStudio project, all of the all the project's pieces are in the same folder. This allows for a clean workflow and a simple working directory for R. When you are executing code for a document/script R will search for things (such as data) in the <u>same</u> folder as the document/script, which is called a *relative path*.

The easiest way to do this is, whenever you start working on a new project in RStudio,

1. Click on the "File" menu button, then select "New Project"

2. Click "New Directory"

3. Click "New Project"

4. Create a name for your project (make it explanatory!)

5. Select where the project should live

6. Click the "Create Project" button

7. Open the project!

After this process `R` will be searching for objects (such as data) in the <u>same</u> folder as the project. This allows for us to keep all of our files in a self-contained system.

Working in `R`

RStudio allows for you to execute commands directly from the code chunk in the document by using the **Ctrl + Enter** (on Macs, **Cmd + Return**) shortcut. If you place your cursor on the line in the code chunk that you would like to run and hit this shortcut, R will execute that line(s) of code for you. Alternatively, you can also execute code in the console (where the output of the commands pops up). The difference between running code in the console and in the document is that any code you execute in the console will be lost once you close your `R` session. If you type code into the document's code chunks, it will be saved when you close your `R` session. Because we want to be able to go back and re-run our code after today's workshop, it is better to type the command we want `R` to run in the document and save it!

If R is ready to accept commands, the `R` console (in the bottom-left) will show a `>` prompt. When `R` receives a command (by typing, copy-pasting, or using the shortcut), it will execute it, and when finished will display the results and show the `>` symbol once again. If `R` is still waiting for you to provide it with additional instructions, a `+` will appear in the console. This should tell you that you didn't finish your command. You could have forgotten to close your parenthesis or a quotation. If this happens and you are unsure of what went wrong, click inside the console and hit the `Esc` key. Then you can start over and figure out where you went wrong!

Calculator

Practice: Enter each of the following commands and confirm that the response is the correct answer.

```r
1 + 2

16 * 9

sqrt(2)

20/5

18.5 - 7.21

3%%2   ## what is this doing?
```

Creating Objects

These operations, however, are not very interesting. To do more useful things in `R`, we need to assign values to an object. To create an object, we tell `R` the object's name, followed by an assignment arrow (`<-`), and finally the value of the object. This would look something like this:

```
x <- 6
```

Once we execute/run this line of code, we notice that a new object appears in our environment window. This window shows all of the objects that you have created during your `R` session. The value of `x` appears next to it, since it is a scalar.

Remarks:

- In the above code `<-` is the assignment operator. It assigns values on the right to objects on the left. So, after executing `year <- 6`, the value of `year` is `6`. The arrow can be read as `6` goes into `year`. For historical reasons, you can also use = for assignments, but not in every context. Because of the slight differences in syntax, it is best practice to always use `<-` for assignments.
    - In RStudio, typing Alt at the same time as the - key will write `<-` in a single keystroke. Neat!
- There are a few simple rules that apply when creating the same of a new object (like we did above):
    - `R` is case sensitive, so if you name your variable `cat` but then try to run the code `Cat + 2`, you will get an error saying that `Cat` does not exist
    - You also want your object's name to be explanatory, but not too long. Think `current_temperature` verses `current_temp`. Do you really want to type out temperature every time?
    - Finally, you cannot begin any object's name with a number. You can end a name with a number (e.g. clean_data2), but does that give you much information about what is in the contents of `clean_data2` relative to `clean_data`?
    - The name cannot contain any punctuation symbols, except for `.` and `_` (`.` is not recommended)
    - You should not name your object the same as any common functions you may use (`mean`, `sd,` etc.)
- Using a consistent coding style makes your code clearer to read for your future self and your collaborators. In `R`, three popular style guides are Google's (https://google.github.io/styleguide/Rguide.html), Jean Fan's (https://jef.works/R-style-guide/) and the tidyverse's (https://style.tidyverse.org/). The tidyverse's is very comprehensive and may seem overwhelming at first, but you'd be surprised what good code writing habits can do for your collaborations!

Clean Code

Yes, writing code may be completely new to you, but there is a difference between code that looks nice and code that does not. Generally, object names should be nouns and function names should be verbs. It is also important that your code looks presentable, so that a friend/college/professor can read it and understand what you are doing. For these reasons, there are style guides for writing code in `R`. The two main style guides are Google's (link) and the slightly more comprehensive Tidyverse style guide (link). Optionally, you can install `lintr` to automatically check and correct for issues in your code styling. More on packages to come!

Working with Objects

When you assign a value to an object (like we did previously) `R` does not output anything by default. If you enclose the code you wrote in parenthesis, then `R` will output the value of the object you created.

```
(x <- 6)
```

```
## [1] 6
```

Once the object has been created, you can use it! Run the following lines of code:

```
2.2 * x
```

```
## [1] 13.2
```

```
4 + x
```

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

```
x%%3
```

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

We can also overwrite an object's value, so that it has a new value. In the code below create a new object y and then we give x a new value of 2.

```
y <- x + 6

x <- 2
```

Exercise 1: What is the current value of y? 12 or 8?

Working with Different Data Types

A vector is the basic data type in R. A vector is a series of values, which can be either numbers or characters, but every entry of the vector must be the same data type. R can tell that you are building a vector when you use the c() function, which concatenates a series of entries together.

```
temps <- c(50, 55, 60, 65)
temps
```

```
## [1] 50 55 60 65
```

To make a vector of characters, you are required to use quotation marks (" ") to indicate to R that the value you are using is not an object you already created in R.

```
animals <- c("cat", "dog", "bird", "fish")
animals
```

```
## [1] "cat"  "dog"  "bird" "fish"
```

Important features of a vector is the type of data they store. Run the following lines of code and decide what type of data the vectors contain.

```
class(temps)
```

```
class(animals)
```

Exercise 2: Create a vector, named dec, that contains decimal valued numbers. Then check what data type does that vector contain?

```
# Exercise 2 code here!
```

Another possible data type is a logical (Boolean) value. This type of data
takes on values of `TRUE` and `FALSE`. But, we said that vectors could only be numbers or characters. If `TRUE`
and `FALSE` don't have quotations around them, then they aren't characters. So, then they must be numbers.
What numbers do you think they are?

```
logic <- c(TRUE, FALSE, FALSE, TRUE)

class(logic)

## [1] "logical"
```

Exercise 3: What happens when we try to mix different data types into one vector? Speculate what will
happen when we run each of the following lines of code:

```
num_char <- c(1, 2, 3, "a")

num_logic <- c(1, 2, 3, FALSE)

char_logic <- c("a", "b", "c", TRUE)

guess <- c(1, 2, 3, "4")
```

In each of these vectors, the two types of data were *coerced* into a single data type. This happens in a
hierarchy, where some data types get preference over others. Can we draw a diagram of the hierarchy?

Lists

While the elements of vectors have to be of the same data type, a list is a special vector in `R` that allows for
you to store a variety of types of objects. If you have a vector, a matrix, a character, you can store all of
them into one list object!

The arguments to the list function are the components of the list. Where the components can be characters,
vectors, matrices, or other data structures. Here, we create a list whose components are the three vectors
we've been working with:

```
my_first_list <- list(animals, temps, logic)
my_first_list

## [[1]]
## [1] "cat"  "dog"  "bird" "fish"
##
## [[2]]
## [1] 50 55 60 65
##
## [[3]]
## [1]  TRUE FALSE FALSE  TRUE
```

We notice that when printing a list, the output looks a bit different. There are a whole bunch of brackets!
Let's break them down. I like to think of a list as a shelf with cubby holes. The cubby holes are the
components of the list, but there are elements in each cubby.

- To get to a specific component (cubby) of a list, you use the double brackets next to the name of the
  list (e.g `my_first_list[[1]]`).

- To access the elements inside each cubby, you then use single square brackets (e.g. `my_first_list[[1]][2]`).
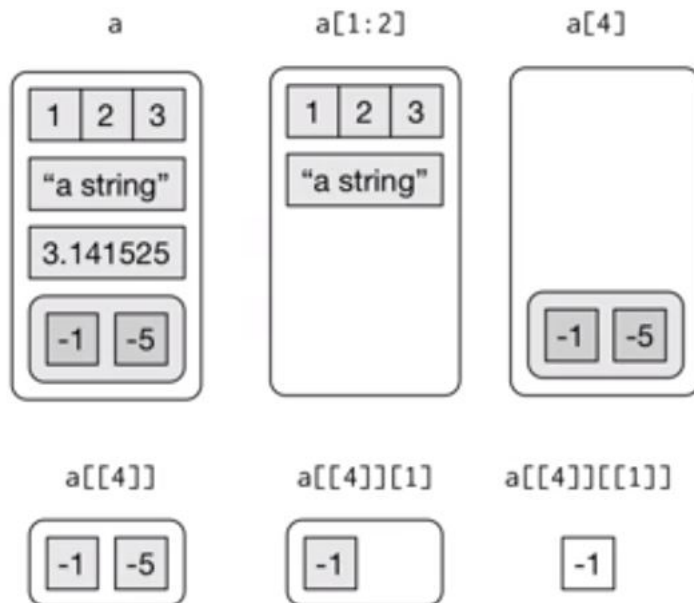
Named Lists

```
my_named_list <- list(title = "statistics", numbers = 1:10, data = TRUE)
my_named_list
```

```
## $title
## [1] "statistics"
##
## $numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $data
## [1] TRUE
```

We see the output for a named list is slightly different. Instead of double brackets next to each component, there are now **$** and names of the variable. This will help you understand the structure of data.frames (coming up next)!

Importing Data

- Use the **Import Dataset** button in the **Environment** tab

- Choose the **From CSV** option

- Click on the **Browse** button

- Direct the computer to where you saved the BlackfootFish data file, click **open**

- It will bring up a preview of the data
- Uncheck the box labeled "Strings as factors"

- Click on the **Import** button

Notice the code that outputs in the console (the bottom left square). This is the code that you could have typed in the code chunk below to import the data yourself. Copy and paste the code that was output in the code chunk below.

```
# copy and paste the code that was used by R to import the data be careful to
# only copy the code that is next to the > signs!
```

Structure of Data

The data we will use is organized into data tables. When you imported the BlackfootFish data into RStudio was saved as an object. You are able to inspect the structure of the BlackfootFish object using functions built in to `R` (no packages necessary).

Run the following code. What is output from each of the following commands?

```r
class(BlackfootFish)  ## What is the object class of the data?
```

```
## [1] "data.frame"
```

```r
dim(BlackfootFish)  ## What is the first number represent? What about the second number?
```

```
## [1] 18352      7
```

```r
names(BlackfootFish)  ## What does this output refer to?
```

```
## [1] "trip"    "mark"    "length"  "weight"  "year"    "section" "species"
```

```r
str(BlackfootFish)  ## What is the structure of the data?
```

```
## 'data.frame':    18352 obs. of  7 variables:
##  $ trip   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ mark   : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ length : num  288 288 285 322 312 363 269 160 213 157 ...
##  $ weight : num  175 190 245 275 300 380 170 40 80 35 ...
##  $ year   : int  1989 1989 1989 1989 1989 1989 1989 1989 1989 1989 ...
##  $ section: chr  "Johnsrud" "Johnsrud" "Johnsrud" "Johnsrud" ...
##  $ species: chr  "RBT" "RBT" "RBT" "RBT" ...
```

```
summary(BlackfootFish)  ## What is the data type of each variable in our dataset?
```

```
##       trip            mark            length           weight
##  Min.   :1.000   Min.   :0.00000   Min.   : 16.0   Min.   :   0.0
##  1st Qu.:1.000   1st Qu.:0.00000   1st Qu.:186.0   1st Qu.:  65.0
##  Median :2.000   Median :0.00000   Median :250.0   Median : 150.0
##  Mean   :1.501   Mean   :0.09285   Mean   :262.3   Mean   : 246.2
##  3rd Qu.:2.000   3rd Qu.:0.00000   3rd Qu.:330.0   3rd Qu.: 330.0
##  Max.   :2.000   Max.   :1.00000   Max.   :986.0   Max.   :4677.0
##                                                    NA's   :1796
##       year          section           species
##  Min.   :1989   Length:18352      Length:18352
##  1st Qu.:1991   Class :character   Class :character
##  Median :1996   Mode  :character   Mode  :character
##  Mean   :1997
##  3rd Qu.:2002
##  Max.   :2006
##
```

```
typeof(BlackfootFish)  ## What is the storage mode of a data.frame?
```

```
## [1] "list"
```

When we inspect dataframes, or other objects in `R`, there are some general functions that are useful to check the content/structure of the data. Here are some:

- size:
  - `dim(data)`: rows and columns

  - `nrow(data)`: number of rows

  - `ncol(data)`: number of columns
- content:
  - `head(data)`: first 6 rows

  - `tail(data)`: last 6 rows

  - `View(data)`: opens viewer window in separate tab
- names:
  - `colnames(data)`: column names of dataframe

  - `rownames(data)`: row names of dataframe
- summary of content:
  - `str(data)`: structure of object and information about the columns

  - `glimpse(data)`: similar information to `str`, but neater output

  - `summary(data)`: summary statistics for each column

Note: Many of these are generic functions that will work on other data types, not just on `data.frame`s.

Dataframes

What is a dataframe? A dataframe is a type of `R` object and is the *de facto* structure of tabular data. You can create dataframes by hand, but most of us do not use `R` to input our data by hand. Instead, we import our data using `R` commands that read in spreadsheets (`read.csv`, `read_csv`, `read_excel` etc.). A dataframe is a set of columns, where each column is a vector. Thus, columns have the same data type *within* the column, but potentially different data types *across* columns.

For example, the columns `trip`, `mark`, and `year` are integers (whole numbers), `weight` and `length` are numeric (numbers with decimals), and `section` and `species` are characters.

Extracting Data

If we were interested in accessing a specific variable in our dataset, we can use the `$` command. This command extracts the specified variable (on the right of the `$` sign) from the dataset. When this is extracted, `R` views the variable as a vector of entries, which is what the [1:18352] refers to.

```
years <- BlackfootFish$year
## extracts year from the dataset and saves it into a new variable named year

str(years)  ## using the new variable (remember case matters!)

##  int [1:18352] 1989 1989 1989 1989 1989 1989 1989 1989 1989 1989 ...
## How would you determine how long the vector is?
```

Another method for accessing data in the dataset is using matrix notation ([row, column]). If you look to your right in the **Environment** window, you notice that RStudio tells you the dimensions of the BlackfootFish data. You can (roughly) view the dataset as a matrix of entries, with variable names for each of the columns. I could instead use bracket notation to perform the same task as above, using the following code.

```
years <- BlackfootFish[, 5]
## This takes ALL rows of data but only the fifth column Same as years <-
## BlackfootFish[1:18352, 5]

str(years)

##  int [1:18352] 1989 1989 1989 1989 1989 1989 1989 1989 1989 1989 ...
```

Practice:

The following is a preview of the dataframe `df`:

```
##   x   y    z
## 1 H May 2010
## 2 N Oct 2015
## 3 T Mar 2018
## 4 W Aug 2017
## 5 V Feb 2019
```

Exercise 4: What would be output if you entered: `df[3, ]`?

Exercise 6: What would you input to get an output of `2015`? Can you think of two ways to do it?

Exercise 7: How would you pull off only columns `x` and `y`? What about pulling off only columns `x` and `z`?

Exercise 8: How would you modify the script below, to get an output of `[1] 22 24`?

```
s <- c(22, 24, 49, 18, 1, 6)
s[]
```

Exercise 9: What would be output if you entered: `s[3, ]`?

Changing Data Type

By default, when building or importing a data frame, the columns that contain characters (i.e. text) are coerced (= converted) into factors. Depending on what you want to do with the data, you may want to keep these columns as character. To do so, read.csv() and read.table() have an argument called stringsAsFactors which can be set to FALSE. This is what happened when we unchecked the "stringsAsFactors" box when importing our data.

In most cases, it is preferable to set stringsAsFactors = FALSE when importing data and to convert as a factor only the columns that require this data type. But what is a factor?

Consider the variables species and section. These variables represent a broader class of what we call categorical variables. In `R` there are two ways to store this information, (1) as a series of `character` strings, or (2) as a `factor`. In the early days of coding in `R`, factors were more efficient than characters, since you only need to store the level of the factor each observation went with.

While factors are still useful in today's statistical analyses and data visualizations, they can be tricky to deal with. When you convert a variable to a factor, for many operations you will get different results than for a character (McNamara & Horton, 2017).

In these data section has two levels (Johnsrud and ScottyBrown) and species has four levels (RBT, WCT, Bull, and Brown). If we want `R` to view these variables as factors instead of characters, we need to change their data type.

```
unique(BlackfootFish$species)  ## tells you the unique values of species
```

```
## [1] "RBT"   "WCT"   "Bull"  "Brown"
```

```
unique(BlackfootFish$section)  ## tells you the unique values of section
```

```
## [1] "Johnsrud"   "ScottyBrown"
```

```
BlackfootFish$speciesF <- as.factor(BlackfootFish$species)
## creates a new variable that is the factor version of species
BlackfootFish$sectionF <- as.factor(BlackfootFish$section)
## creates a new variable that is the factor version of section
```

There is also a function that will allow for you to specify the order of the levels of a factor! As we saw before, the `as.factor` function chooses the levels alphabetically. Suppose you would like for the species to be in the following order: Bull, Brown, RBT, and WCT.

Using the `factor` function this would look like:

```r
BlackfootFish$speciesF <- factor(BlackfootFish$species,
                                 levels = c("Bull", "Brown", "RBT", "WCT"))
```

Practice:

Exercise 11: Year was saved as an integer data type (1989 - 2006), but we may want to consider it to be a categorical variable (a factor). Write the `R` code to create a new variable called `yearF` that is a factor of `year` (as you did with section and species).

Exercise 12: Now, verify that `yearF` is viewed as a categorical variable, with the same levels as `year`. (hint: you have already used three functions that would do this for you)

The issue with factors, lies with if/when you want to change it back to a number or character. In the code below I've decided that I don't want year to be a factor and want to change it back to numeric. What happens when I use the `as.numeric()` function on the yearF variable?

```r
year_recover <- as.numeric(BlackfootFish$yearF)

ds <- data.frame(original = BlackfootFish$yearF,
                 recovered = year_recover)
head(ds)
```

```
##   original recovered
## 1     1989         1
## 2     1989         1
## 3     1989         1
## 4     1989         1
## 5     1989         1
## 6     1989         1
```

Packages

As we mentioned previously, `R` has many packages, which people around the world work on to provide and maintain new software and new capabilities for `R`. You will slowly accumulate a number of packages that you use often for a variety of purposes. In order to use the elements (data, functions) of the packages, you have to first install the package (only once) and then load the package (every time).

We're going to install a few packages that are often used.

- Use the **Install** button in the **Packages** tab

- Type in `devtools` and `tidyverse` into the blank line (separated by a comma)

- Check the **Install dependencies** box

- Click on the **Import** button

12

There will be a large amount of output coming out of the console. This output is `R` trying to download the package(s) you requested by contacting the mirror you chose for it to use when downloading (I chose Northern Michigan University). Once the computer has downloaded the packages, it will tell you that "`The downloaded binary packages are in`", followed by the location of the files.

Now that the files are downloaded, we need to load them in order to use them. The following code will load each package, please run it!

```
library(devtools)
```

```
## Loading required package: usethis
```

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------- tidyverse 1.3.0 --
## v ggplot2 3.2.1     v dplyr   0.8.3
## v tibble  2.1.3     v stringr 1.4.0
## v tidyr   1.0.0     v forcats 0.4.0
## v purrr   0.3.3
## -- Conflicts ------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Notice that when loading the `tidyverse` package that there is a large amount of output. This output is telling you all of the other packages that are loaded in the `tidyverse` package, as well as the functions in the `tidyverse` package that overwrite (mask) functions from base `R`.

This is the process you go through if you ever find packages that you would like to use! Often packages that you install will need to be updated. To update a package you can click on the "Tools" tab, then click on "Check for Package Updates". This will bring up a window that will list all of the packages that have newer versions than what you have. Click on the packages that you wish to update, or click on the "Select All" button.

Finding Help

One of the chief reasons for `R`'s religious following is its wonderful documentation. If you know a function does what you want (say find the variance), but are not quite sure how it's spelled, what arguments it takes, or what package it lives in, don't fret! The `?` and `help()` commands are very powerful. For functions, placing the `?` before the name, will tell `R` to search for that name in all of the functions, in all of the packages you have installed.

- If it finds *one* **identical match**, it will display the help file for that function in the Help tab in the bottom-right corner.

- If it finds *more than one* **identical match**, it will display the functions, in their respective packages, that you have to choose from.

- If it find *no* **identical match**, it will tell you that "`No documentation  for ____ in specified packages and libraries:,`" and suggests you use a `??` instead.
  - A `??` in front of the function name will search **all of `R`** for named functions similar to what you typed.

– The output will tell you what package the function is in, as well as the function's name (`package::function`).

If you would like help on a particular package, say one that you just downloaded, then you can use the same command(s) to get help. These commands will load up a help page (in RStudio) in the Help pane. Each help page is broken down into sections:

- Description: An extended description of what the function does.

- Usage: The arguments of the function and their default values.

- Arguments: An explanation of the object each argument is expecting.

- Details: Any important details to be aware of.

- Value: The object the function returns.

- See Also: Any related functions that may be useful.

- Examples: Some examples for how to use the function.

Functions

In `R` there are both functions that are built in (require no package to be loaded), as well as functions that are housed within specific packages. You have already used a few built in functions to inspect the structure of the BlackfootFish data (`str`, `class`, `summary`). As we know, a function transforms an input (potentially multiple) into an output. You have to provide `R` with the inputs (arguments) required for the function to generate an output. The argument(s) inside a function happen after the `(` symbol. You know an object is a function when it is immediately followed by a `(` and the corresponding closing `)` comes after the arguments are complete. The output of a function does not have to be numerical and it typically is not a single number, it can be a set of things or a dataset.

Arguments describe the details of what a function is to do. Some functions take arguments that are specified by the user, or, if left undeclared, take on default values. These arguments are typically given names (as seen in the help file), but the arguments are assumed to follow the order the function expects if they are not named (also stated in the help file). When naming an argument, the name of the argument is followed by an `=` sign and then the value of the argument. Notice that here we are using the `=` to declare what value each argument is taking on, we **are not** creating a new variable with that value assigned to it.

Suppose we wanted to create a vector of 10 zeros. To do this, we would use both the `rep` function:

```
# ?rep

rep(0, times = 10)  ## repeating 0 three times

##  [1] 0 0 0 0 0 0 0 0 0 0
rep(times = 10, 0)  ## switching order of arguments

##  [1] 0 0 0 0 0 0 0 0 0 0
rep(0, 10)  ## no named arguments

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

```
rep(10, 0)   ## not what we wanted!
```

numeric(0)

Now let's look over some other functions that are often used:

```
mean(BlackfootFish$weight)  ## takes a numerical input, but there are NA's in our data

mean(BlackfootFish$weight, argument here! )  ## add in the argument that removes the NA's

median(BlackfootFish$species) ## gives an error because the input is not the correct data type

cor(BlackfootFish$length, BlackfootFish$weight) ## takes multiple inputs separated by a comma

## Does cor have an option to remove NA's?
```

As seen in the functions above, some functions have *optional* arguments. If they are not specified by the user, then they take on their default value (`FALSE` for `na.rm`). These options control the behavior of the functions, such as whether it includes/excludes NA values.

Cleaning Data

In many instances, you will deal with data that are not "clean". Based on the output we received from the `mean()` function, we know that there are NA's in the BlackfootFish data, possibly across a variety of variables. Before we used `na.rm` as an option to remove NA's *within* a function, but the `na.omit` function takes a dataframe and removes any NA's from that dataset. Based on the output below, how many rows in the BlackfootFish data have an NA present?

```
dim(BlackfootFish)  ## gives the dimensions of the dataset in (row, column) format
```

[1] 18352   10

```
dim(na.omit(BlackfootFish))
```

[1] 16556   10
na.omit takes dataframes, matricies, and vectors and returns the object with
incomplete cases removed (NA's removed)

Remark: The computer is using an algorithm to return a dataset with no NA values anywhere in it. This algorithm goes through every row of the dataset and (roughly) has the following steps,

- Inspect the row to see if there is an NA anywhere in that row

- If there is an NA in that row, the logical (`is.na`) evaluates to TRUE, and the row is deleted

- If there is not any NA's in that row, the logical evaluates to FALSE, and the row is retained

- Once it has stepped through every row, the function outputs the "cleaned" dataframe

Subsetting Data

If we wish to remove all of the NA's from the dataset, we can simply use the `na.omit` command from above. We can save the new "clean" dataset under a new name (creating a new object) or under the same name as before (replacing the old object with the new object).

```
BlackfootFish_clean <- na.omit(BlackfootFish)
## Creates a new dataframe, where the NA's have all been removed
```

Data Visualization

There are many different genres of data graphics, with many different variations on each genre. Here are some commonly encountered kinds:

- **scatterplots**: showing relationships between two quantitative variables

- **distributions**: showing distributions of a single quantitative variable

- **bar charts**: displaying frequencies or densities of a single categorical variable

Scatterplots

The main purpose of the scatterplot is to show the relationship between two variables across several or many cases. Most often, there is a Cartesian coordinate system in which the x-axis represents one variable and the y-axis the second variable.

```
#?plot()
plot(length ~ weight,
     data = BlackfootFish_clean)
```
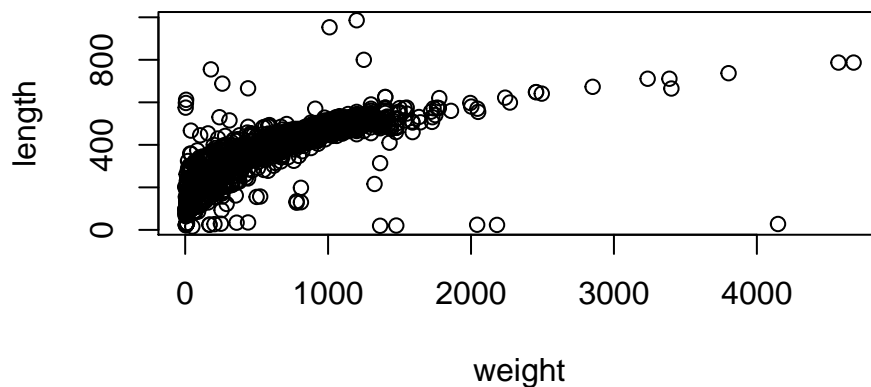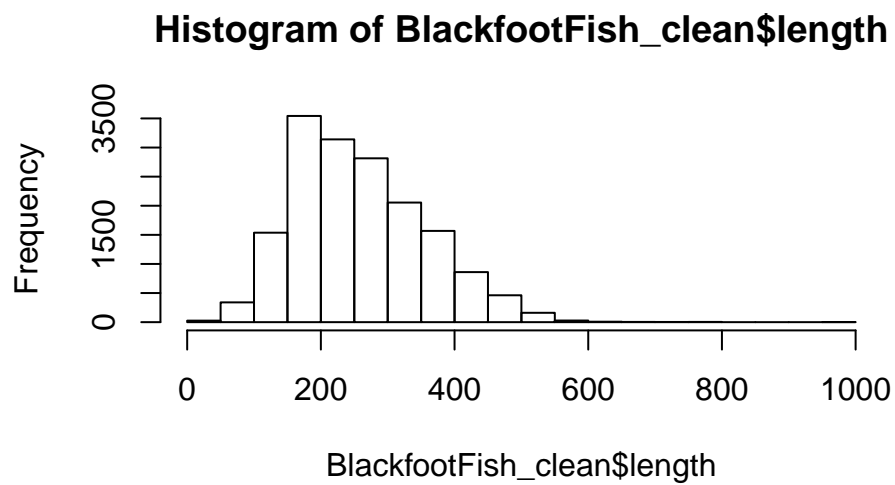
```
plot(length ~ weight,
    data = BlackfootFish_clean,
    xlab = "Weight (gm)", ## adding in axis labels
    ylab = "Length (cm)",
    las = 1 ## changing orientation of axis labels
    )
```
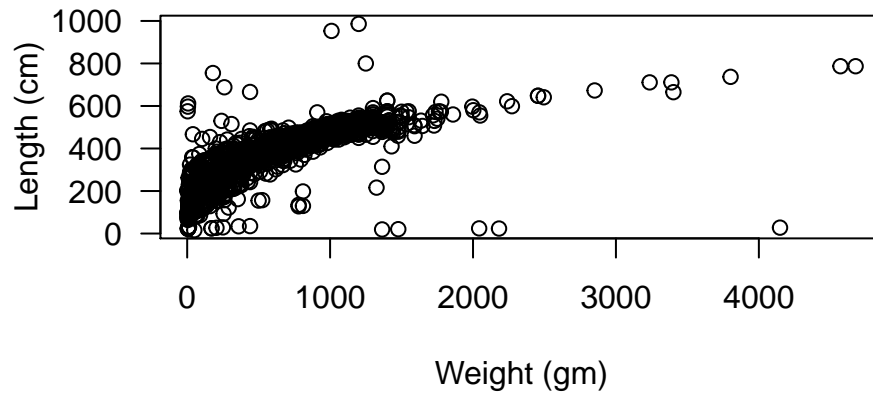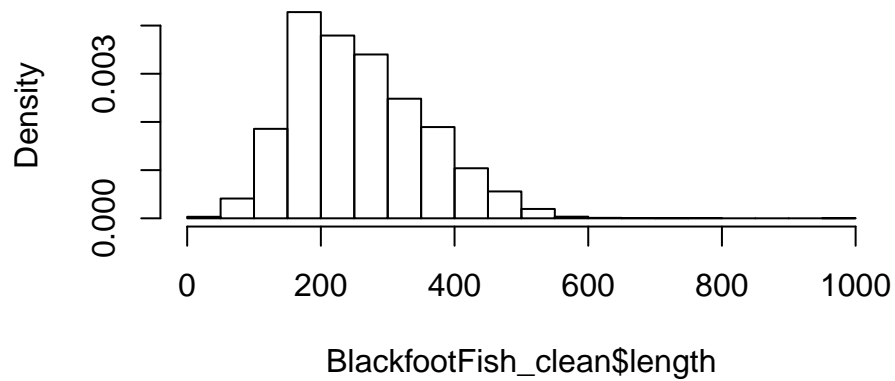

Distribution

A histogram shows how many observations fall into a given range of values of a variable.

```
hist(BlackfootFish_clean$length)
```

```
hist(BlackfootFish_clean$length,
     freq = F) ## converts to a density plot (area adds to 1)
```
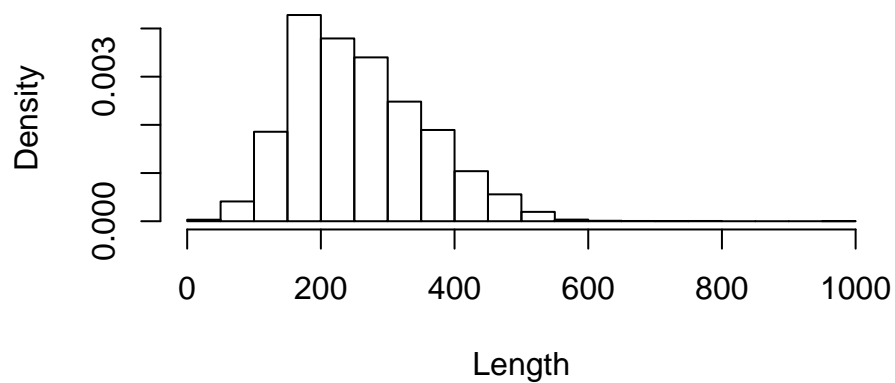
Histogram of BlackfootFish_clean$length


```
## Does freq need to be named?
hist(BlackfootFish_clean$length, FALSE)
## Why is there an error about the 'number of breaks'?
```

```
hist(BlackfootFish_clean$length,
     freq = F,
     xlab = "Length", ## adds x-axis label
     main = "Fish Lengths in Blackfoot River" ## adds title to plot
     )
```

Fish Lengths in Blackfoot River

```r
hist(BlackfootFish_clean$length,
    freq = F,
    nclass = 50, ## changes the number of bins
    xlab = "Length",
    main = "Fish Lengths in the Blackfoot River",
    las = 1 ## changes orientation of axis labels
    )
```
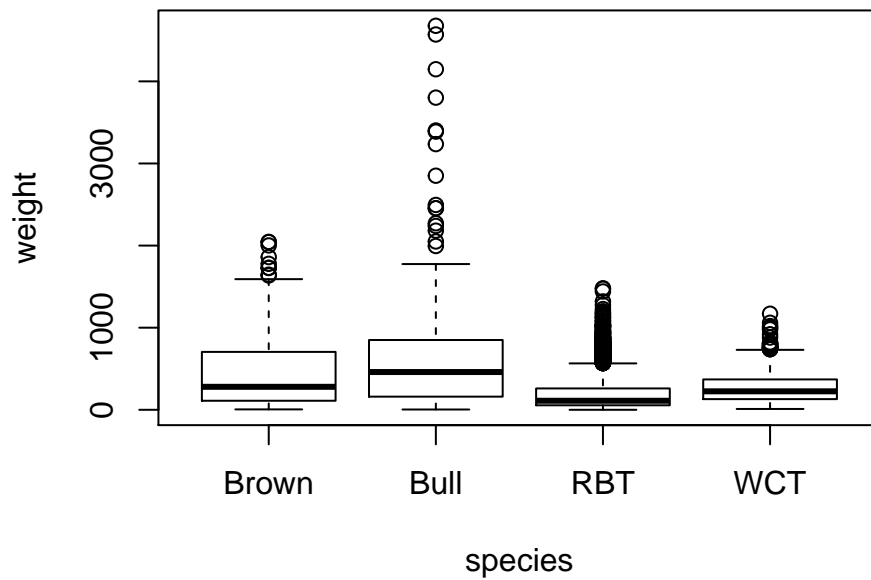
Fish Lengths in the Blackfoot River
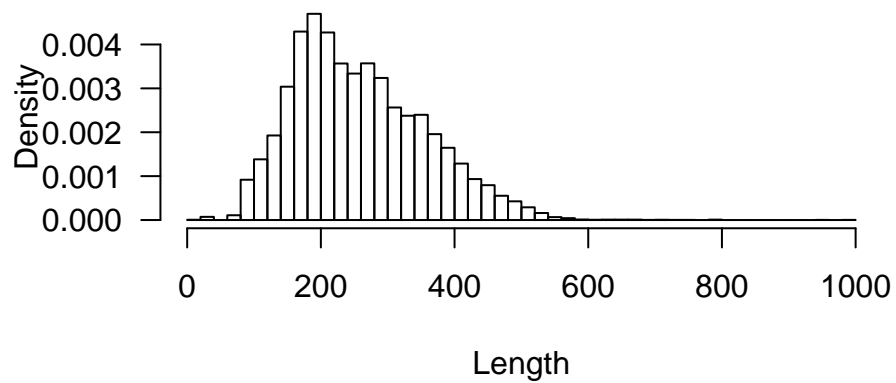
Side-by-Side Boxplots

The familiar boxplot is effective when the objective is to compare the distribution of a quantitative variable across different levels of a categorical variable.

```r
## What other options are available to add to your boxplot?
boxplot(weight ~ species,
        data = BlackfootFish_clean)
```

Bar Charts

Bar charts are an effective way to compare the frequencies of levels of a categorical variable.

```r
section <- table(BlackfootFish_clean$section)
## tables the number of fish that were caught in each section

barplot(section,
        xlab = "Section", ## adds axis labels to plot
        ylab = "Number of Fish",
        main = "Fish Caught by Section", ## adds title to plot
        las = 1, ## changes orientation of axis labels
        col = "blue", ## specifies what color the bars should be
        ylim = c(0, 12000) ## specifies what range of y-values to plot
        )
```

Fish Caught by Section
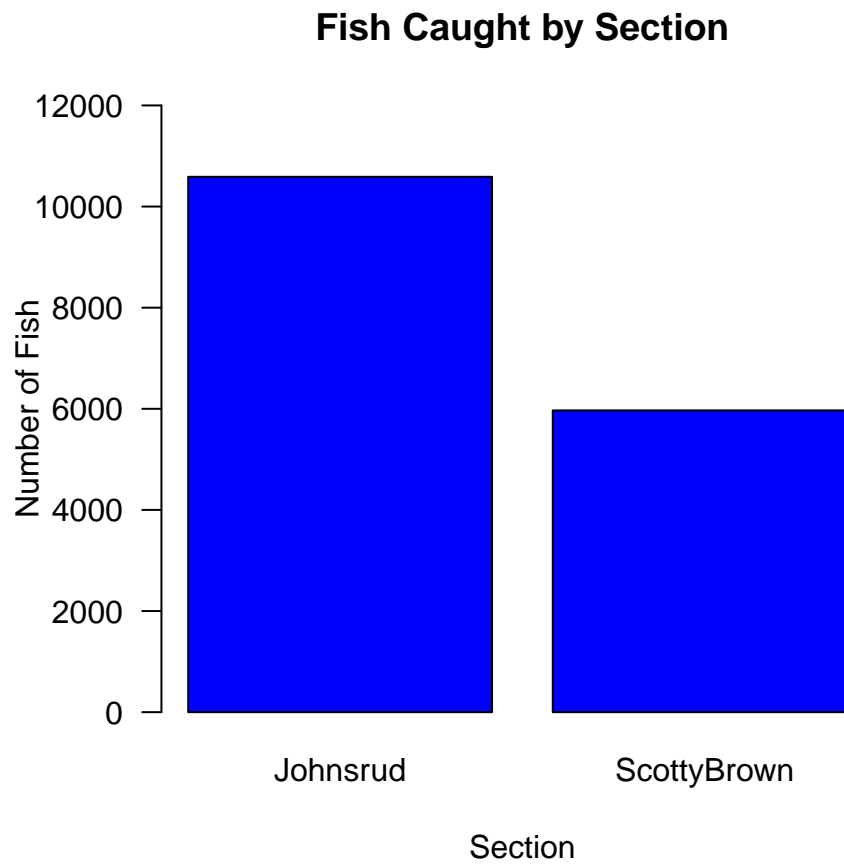
Practice

Exercise 13: Using statistics or graphics, which year in our dataset had the most fish caught?

Exercise 14: Make a boxplot of the fish weights over the different years in the dataset.

Terminology Used in Workshop

- *Command*: A command is what `R` executes. In an `R` script file (script.R), commands are automatically implied, as this type of file does not accept text, only in comments. In an Rnw (Markdown) file (report.Rnw), commands are delineated between three ticks on the top (`{r})` and three ticks ()` on the bottom.

- *Comment*: Helpful text added into a script environment. Comments can be used to describe functions, processes, a train of thought, so that when you return to your code, tomorrow or next year, you are able to understand the purpose of each line of code!

- *Object*: A variable created in `R`, to be used elsewhere in the code. Objects can be a variety of things, such as scalars (x <- 3), vectors (x <- c(1, 2, 3, 4, 5)), matrices, and dataframes, to name a few.

- *Assignment Arrow*: The assignment arrow `<-` is used to assign values on the right to the objects on the left (x <- 1). For historical reasons, you can also use `=` for assignments, but not everywhere. Because of these slight differences, it is recommended to *always* use assignment arrows for assignment.

- *Class*: Most `R` objects have a class attribute, a character vector giving the names of the classes from which the object inherits. Examples of classes are numeric, factor, integer, character, dataframe, matrix, list.

- *Vector*: A vector is a list of entries, all sharing the same class. A vector has only one dimension, so data extraction uses only a single entry in brackets (e.g. x[3]). You can create vectors of characters (c("a", "b", "c")), vectors of numbers (c(1, 2, 3)), to name a few.

- *Matrix*: Similar to what you may have seen in a mathematics class, a matrix is an object with rows and columns, where every entry in the matrix must be a number.

- *List*: A generic vector, which contains other objects. A list can contain a variety of different classes of objects, e.g. characters, vectors, data.frames, matrices, or outputs from a model! A data.frame is a special type of list where the components are vectors and they all have the same length.

- *Dataframe*: A dataframe is a collection of variables. Dataframes share many of the properties of matrices, where you are able to extract elements using bracket (`[]`) notation, and lists, where you are able to extract columns using `$`. Dataframes are used as the fundamental data structure by most of `R`'s statistical modeling software.

- *Argument*: Input(s) into a function, so that an output is created. Most functions take named arguments (e.g. data = BlackfootFish) and the order of the arguments is assumed to follow the order found in the function's help file. When using a named argument in a function, the name comes first, followed by an `=` sign, then the input.

- *Logical Value*: TRUE and FALSE value(s) that can be used to turn off/on options in functions and plots, and also to manipulate data.

Workshop Materials & Recordings Available:

- through RStudio Cloud at: http://bit.ly/introduction-R

- through Allison's personal website at: http://www.math.montana.edu/allison_theobold

- through the MSU Library YouTube channel: https://www.youtube.com/watch?v=W6E3hpcoUkQ& feature=youtu.be

How to Learn More About `R`

This material is intended to provide you with an introduction to using `R` for scientific analyses of data. The best way for you to continue to learn more about `R` is to use it in your research! This may sound daunting, but writing `R` scripts is the best way to become familiar with the syntax. This will help you progress through more advanced operations, such as cleaning your data, using statistical methods, or creating graphics.

The best place to start is playing around with the code from today's workshop. Change parts of the code and see what happens! Better yet, use the code from the workshop to investigate your own data!

Workshop build on: 2020-01-21 13:09:23

Instructor Biographies

Allison Theobold

Allison Theobold, originally from Grand Junction, CO, is a sixth-year graduate student studying Statistics Education at Montana State University. Allison holds a Master's degree in Statistics from Montana State University, and will be defending her dissertation in April of this year. Recognized by both the Department of Mathematical Sciences and the College of Letters and Sciences as an outstanding graduate researcher, Allison's passion for teaching statistical computing is infectious. Allison has taught Introduction and Intermediate Statistics to undergraduate and graduate students, while also completing two years of statistical consulting for diverse groups at Montana State. Her experiences as a teacher and consultant established an interest in preparing researchers in the sciences with the computational tools necessary to implement statistics.

Elijah Meyer

Elijah was born and raised in Great Falls, Montana and currently holds a Master's degree in Statistics from Montana State University. His early research involved sports statistics, focusing in on Fitbit data and Disc Golf visualizations. His recent work includes research on Graduate Teaching Assistants (GTAs) and program development to help better support GTAs when teaching statistics. Many of these projects were presented to audiences at the Joint Statistical Meetings, Cascadia Symposium of Statistics in Sports, and United States Conference on Teaching Statistics 2019. He is currently pursuing a Ph.D. in statistics with a focus on statistics education.

Kelly Loucks

Kelly is from Charlotte, North Carolina and is a second-year Master's student studying Statistics. She previously earned her Bachelor's degree in Mathematics with a concentration in Statistics from Appalachian State University. She has worked on research for the United States Forest Service and taught Introduction to Statistics. She enjoys skiing, hiking, ice climbing, and mountain biking in her free time.