Data Cleaning and Manipulation

Cole & Emma

Fall 2020

A quick note to contextualise the content we cover today: When we first began discussing running this workshop series, we had a series of intense discussions that were half serious debate, half friendly bickering, about how to teach this particular section. The two of us come from very different learning backgrounds in terms of the R programming environment, and we both currently work with data (data cleaning and management) in very different ways. These two philosophies of working with data in R can be referred to as 'BaseR' and 'TidyR' approaches to working with data. For clarity, the philosophy of the BaseR approach is to use built-in functions and methods to perform operations on data, with syntax that follow the original R programming mechanisms. The TidyR approach on the other hand, relies completely on a set of packages that together are known as the 'Tidyverse', primarily produced and maintained by Hadley Wickham and his colleagues at RStudio. While both of these philosophies/methods sets are perfectly functional and perform the same general set of operations, they differ greatly in their syntax, the types of tools they use, and the general philosophy behind those tools. After much discussion, we agreed that, while the TidyR approach is (debatably) easier to learn, and syntactically more consistent and easy to read, it is inefficient, often inoperable with base R commmands, and untranslatable to any other programming language. Learning how to do things in Base R allows you to access the wide world of R programming in the way it was built to do statistics. Learning how to do things in the Tidyverse is, truly, like learning a completely different programming language to Base R, and allows you access relatively little base content. Thus we concluded it is much more important to learn base R first, as without an appropriate understanding of BaseR functionality, it's hard to get the most out of R when you become more advanced. This whole TidyR vs. BaseR debate is related to the note I've included at the end of this document about using packages. If you're interested in a further treatment of the topic of packages and the TidyR vs. BaseR debate, please feel free to peruse my musings on that topic at your leisure.

Why are we telling you this? We think it's important to give you our motivation for why we're teaching you this way. The majority of online tutorials and how-to's etc. that you may find when Googling R problems related to data manipulation, management, or cleaning will be taught in the Tidyverse. This is because the creators of the Tidyverse (the for-profit company, RStudio) invest a HUGE amount of time and money into education and outreach, particularly outreach to try to push teaching the Tidyverse as the way to go for data management in R. You will likely come across this great wealth of online content and wonder why we didn't teach you those methods. A fair question that we hope is at least partially addressed here and in the continuation of this topic at the end of this document.

Now on to the session...

Why clean data?

We cannot begin a discussion about data cleaning without talking about data themselves. What are data? Well, according to the Oxford Dictionary, data are:

> da?ta (noun) -- facts, and statistics collected together for reference or analysis

That's obviously a very broad definition of what data are, so for our purposes, let's think of data as items of information regarding a particular aspect of the world around us that, when carefully collected, analyzed, and visualized, can tell us something about the system we're studying.

So why do we need to clean it? Has it fallen in some mud? The process of 'cleaning' data is required because the data we collect or receive from other sources are very rarely ready to plug directly into an analysis. They require re-formatting, sub-setting, checking for errors, etc. Strong coding skills (in this case, in R) can make the process of preparing your data for analysis more efficient, less painful, and easier to reproduce. Often, the data cleaning step takes more time and effort than the analysis itself.

What is involved in cleaning data?

The process of preparing entered data for analysis includes (but is not limited to): 1. Removing unwanted observations (e.g., data a trial that went wrong). 2. Handling missing data (e.g., choosing whether or not to impute data for missing values). 3. Managing outliers (e.g., finding and determining whether outliers are biological or due to human error). 4. Fixing structural errors (e.g., ensuring all values for a character variable are consitently capitalised). 5. Re-formatting or transforming dataframes (e.g., transposing from wide to long format). 6. Merging datasets (e.g., pulling variables from multiple files into one dataframe)

How to avoid needing to clean data

The process of cleaning data can be eliminated or, more likely, minimized with some careful thought before collecting or requesting data. Investing time in developing a clear research question and even planning your statistical analysis prior to obtaining data can support a more efficient data cleaning process. Specific things to consider include: 1. Plan your analysis before collecting or requesting (e.g., from a government agency) data. 2. Have a systematic process for collecting, entering, and double checking your data to minimise errors. 3. Enter your data in a format that is conducive to your planned analysis (e.g., long vs. wide format) 3. Scan datasheets as you enter data, keep thorough metadata and notes. 4. Find ways to prevent mistakes (e.g., setting Excel cells to only accept a range of biologically plausible values).

How to clean cata

A first, very necessary step is to read in your data. You all should have the CSV file we'll be using today, it's some data from a project Emma and Cole have both worked on, investigating whether sea lice (a parasite that live on salmon) are developing resistance to a common parasiticide used on salmon farms off the west coast of British Columbia.

```
dir = 'C:/Users/ematkinson/Documents/GitHub/Introduction-to-Programming-for-Biology/data'
setwd(dir)
data = read.csv('week_three_data.csv')
```

Before we do any cleaning at all, let's just take a look at a subset of the thing.

```
head(data)
```

```
##    location    region         date slice_conc louse_sex louse_stage temp_0_hrs
## 1 Burdwoods Broughton June 11 2012          0         f          p2          8
## 2 Burdwoods Broughton June 11 2012          0         m          p2          8
## 3 Burdwoods Broughton June 11 2012          0         m          p2          8
## 4 Burdwoods Broughton June 11 2012          0         m          p2          8
## 5 Burdwoods Broughton June 11 2012          0         m          p1          8
## 6 Burdwoods Broughton June 11 2012          0         m          p1          8
##   temp_6_hrs temp_12_hrs temp_18_hrs temp_24_hrs dead moribund live avg_temp
## 1         13           9           8          10    0        0    1      9.6
## 2         13           9           8          10    0        0    1      9.6
## 3         13           9           8          10    0        0    1      9.6
## 4         13           9           8          10    0        0    1      9.6
## 5         13           9           8          10    0        0    1      9.6
## 6         13           9           8          10    0        0    1      9.6
##   dead_and_moribund comments
```

```
## 1                 0
## 2                 0
## 3                 0
## 4                 0
## 5                 0
## 6                 0
```

Okay, so there's some categorical variables in here, some numerical variables by the looks of it. We can't quite see it all though, what are the names of all the varialbes?

```
names(data)
```

```
##  [1] "location"          "region"            "date"
##  [4] "slice_conc"        "louse_sex"         "louse_stage"
##  [7] "temp_0_hrs"        "temp_6_hrs"        "temp_12_hrs"
## [10] "temp_18_hrs"       "temp_24_hrs"       "dead"
## [13] "moribund"          "live"              "avg_temp"
## [16] "dead_and_moribund" "comments"
```

Okay great. Let's take a more detailed look at the structure of the dataframe.

```
str(data)
```

```
## 'data.frame':    374 obs. of  17 variables:
##  $ location         : Factor w/ 7 levels "Burddwoods","Burdwoods",..: 2 2 2 2 2 2 2 2 2 1 ...
##  $ region           : Factor w/ 2 levels "Broughton","Discovery": 1 1 1 1 1 1 1 1 1 1 ...
##  $ date             : Factor w/ 11 levels "July 2 2012",..: 2 2 2 2 2 2 2 2 2 2 ...
##  $ slice_conc       : int  0 0 0 0 0 0 0 30 30 30 ...
##  $ louse_sex        : Factor w/ 2 levels "f","m": 1 2 2 2 2 2 2 1 2 2 ...
##  $ louse_stage      : Factor w/ 3 levels "a","p1","p2": 3 3 3 3 2 2 2 3 3 3 ...
##  $ temp_0_hrs       : num  8 8 8 8 8 8 8 8 8 8 ...
##  $ temp_6_hrs       : num  13 13 13 13 13 13 13 13 13 13 ...
##  $ temp_12_hrs      : num  9 9 9 9 9 9 9 9 9 9 ...
##  $ temp_18_hrs      : num  8 8 8 8 8 8 8 8 8 8 ...
##  $ temp_24_hrs      : num  10 10 10 10 10 10 10 10 10 10 ...
##  $ dead             : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ moribund         : int  0 0 0 0 0 0 0 1 0 1 ...
##  $ live             : int  1 1 1 1 1 1 1 0 1 0 ...
##  $ avg_temp         : num  9.6 9.6 9.6 9.6 9.6 9.6 9.6 9.6 9.6 9.6 ...
##  $ dead_and_moribund: int  0 0 0 0 0 0 0 1 0 1 ...
##  $ comments         : Factor w/ 5 levels "","(Scott asked me to make the below notes)",..: 1 1 1 1 1
```

We get a very detailed print out here telling us the type of all the columns (variables), and giving us a preview of them. Looking at the first few entries in the 'location' column, we notice that there looks like there might be a typo in one of the entries. The 'location' column gives the names of the local sites where sea lice were collected. Let's investigate...

```
# use the unique() function to return all unique values in the 'location' column
unique(data$location)
```

```
## [1] Burdwoods                  Burddwoods
## [3] Wicklow                    Glacier and Wicklow
## [5] Okisollo Ch                Green Sea Bay
## [7] Burdwoods, wicklow, glacier
## 7 Levels: Burddwoods Burdwoods ... Wicklow
```

```
# find out which row contains the typo
data[data$location=="Burddwoods",]
```

```
##      location    region         date slice_conc louse_sex louse_stage
## 10 Burddwoods Broughton June 11 2012         30         m          p2
##    temp_0_hrs temp_6_hrs temp_12_hrs temp_18_hrs temp_24_hrs dead moribund live
## 10          8         13           9           8          10    0        1    0
##    avg_temp dead_and_moribund comments
## 10      9.6                 1
```

```
# rewrite the entry in the 10th row to correct the typo
data$location[10] <- "Burdwoods"

# check to see what class of vector the 'location' column is
class(data$location)
```

```
## [1] "factor"
```

```
# check to see what levels (i.e., categories) are listed for the vector
levels(data$location)
```

```
## [1] "Burddwoods"                  "Burdwoods"
## [3] "Burdwoods, wicklow, glacier" "Glacier and Wicklow "
## [5] "Green Sea Bay "              "Okisollo Ch"
## [7] "Wicklow"
```

Notice that even after we correct the typo, the incorrect site name still shows up as one of the possible categories. We can correct that by re-factoring (in other words, re-setting) the vector in question:

```
data$location <- as.factor(as.character(data$location))

levels(data$location)
```

```
## [1] "Burdwoods"                   "Burdwoods, wicklow, glacier"
## [3] "Glacier and Wicklow "        "Green Sea Bay "
## [5] "Okisollo Ch"                 "Wicklow"
```

Here, in one line of code, we re-format the vector as a character data type and then re-factor the character vector. If we check, we see that the incorrect site name is no longer one of the categories listed.

So, now we have caught and corrected an error in the dataframe. Notice the general process of (1) finding the mistake at a very broad scale, (2) specifically locating the error, (3) correcting the error, and (4) checking to make sure our fix worked. This is the general recipe for most of data cleaning. Lots of finding, fixing, checking, checking again!

Moving on, perhaps we want to subset for just one focal region. These experiments were conducted in two focal regions (the Broughton Archipelago and the Discovery Islands) but we might just want to analyse the data from one region.

```
# Check how many regions there are
unique(data$region)
```

```
## [1] Broughton Discovery
## Levels: Broughton Discovery
```

```
# Create a new object where you will store your subset of the original data
# It is good practice to create new dataframes when you are performing major
# changes so that you still have the original copy in your environment to
# refer back to easily.
d2 <- data[data$region=="Broughton",]

# Double check...
unique(d2$region)
```

```
## [1] Broughton
## Levels: Broughton Discovery
```

```
# And re-factor...
d2$region <- as.factor(as.character(d2$region))
```

Next, to facilitate our analysis, we want to create a new column for trial number which will assign a unique number for each independent trial. Because we know that there was never more than one trial run on a given date in a given region, we can do this by writing a for loop that uses date to assign unique trial numbers.

```
# First, we're going to re-classify the data vector as a 'character' data type
d2$date <- as.character(d2$date)

# We can create a new column called 'trial' and fill it with NAs for now
d2$trial <- NA
head(d2$trial)
```

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

```
# And then we build a loop to re-write those NAs with the appropriate trial number...
for (i in 1:length(unique(d2$date))) { # for each number starting from 1 and going to
                                       # the max number of unique dates in the data frame
    t <- unique(d2$date)[i] # temporarily assign the given date to 't'
    d2[d2$date==t,]$trial <- i # for all rows where the date matches the current date assigned to 't',
                               # overwrite the trial column with the appropriate number
                               # (e.g., the first date will be assigned trial 1)
  } # end loop
```

Now we check to make sure our loop did what we wanted it to do...

```
# For a given date, how many unique trials are there (should only be one)?
unique(d2[d2$date=="June 11 2012",]$trial)
```

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

```
# For a given trial, how many unique dates are there (should only be one)?
unique(d2[d2$trial==7,]$date)
```

```
## [1] "June 13 2015"
```

Beautiful. Next, we want to get rid of one of the columns. We've looked over the 'comments' column and decided it won't be useful for the analysis and is cluttering things up...

```
# Find out which column number the 'comments' column is...
names(d2)
```

```
##  [1] "location"          "region"            "date"
##  [4] "slice_conc"        "louse_sex"         "louse_stage"
##  [7] "temp_0_hrs"        "temp_6_hrs"        "temp_12_hrs"
## [10] "temp_18_hrs"       "temp_24_hrs"       "dead"
## [13] "moribund"          "live"              "avg_temp"
## [16] "dead_and_moribund" "comments"          "trial"
```

```
# Remove the 17th column and assign the resulting dataframe to a new name
d3 <- d2[,-17] # remember, we index data frames as [rows, columns]
# Check...
head(d3)
```

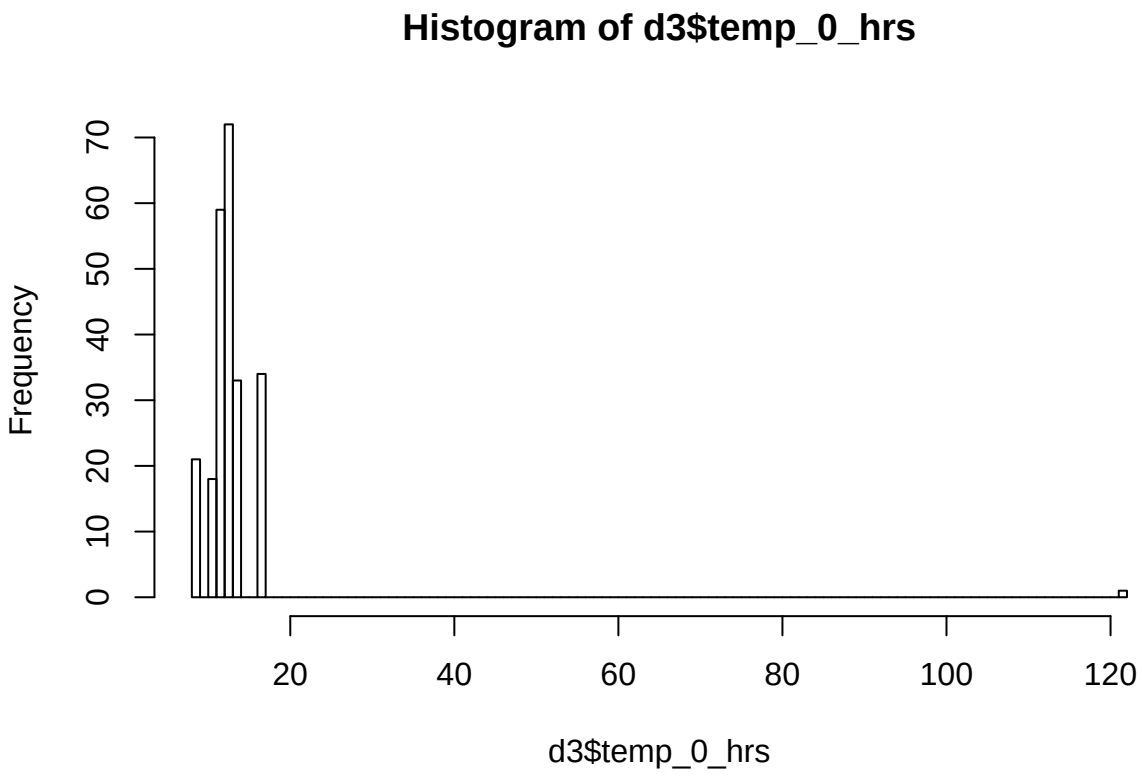
```
##    location    region         date slice_conc louse_sex louse_stage temp_0_hrs
## 1 Burdwoods Broughton June 11 2012          0         f          p2          8
```

```
## 2 Burdwoods Broughton June 11 2012          0         m          p2          8
## 3 Burdwoods Broughton June 11 2012          0         m          p2          8
## 4 Burdwoods Broughton June 11 2012          0         m          p2          8
## 5 Burdwoods Broughton June 11 2012          0         m          p1          8
## 6 Burdwoods Broughton June 11 2012          0         m          p1          8
##   temp_6_hrs temp_12_hrs temp_18_hrs temp_24_hrs dead moribund live avg_temp
## 1         13           9           8          10    0        0    1      9.6
## 2         13           9           8          10    0        0    1      9.6
## 3         13           9           8          10    0        0    1      9.6
## 4         13           9           8          10    0        0    1      9.6
## 5         13           9           8          10    0        0    1      9.6
## 6         13           9           8          10    0        0    1      9.6
##   dead_and_moribund trial
## 1                 0     1
## 2                 0     1
## 3                 0     1
## 4                 0     1
## 5                 0     1
## 6                 0     1
```

Finally, we are going to go over an example of finding and removing an outlier. When it comes to finding errors in your data, it would be extremely time consuming to go row by row looking for outliers. Furthermore, where it was easy to use unique() to look at the list of unique locations and realise that there was an error in one of the "Burdwoods" entries, if we tried to do that with numerical data we'd get an overwhelming number of unique values to dig through. This is where plotting data can come in very handy.

Let's take the example of temperature at the beginning of the trial ('temp_0_hrs'). First, we'll just plot a super simple histogram to look at the distribution of the data.

```
hist(d3$temp_0_hrs, breaks=100) # we use the 'breaks' argument to specify a finer division of bins (i.e
```



It is pretty obvious that while most of the data fall between 10 and 20 degrees celcius, there is a notable outlier above 100 degrees celcius. This is biologically implausible so let's try to track down exactly which trial this was...

```
# Filter for values above 100
d3[d3$temp_0_hrs > 100,]
```

```
##     location    region         date slice_conc louse_sex louse_stage temp_0_hrs
## 55 Burdwoods Broughton June 20 2012         30         f          p2        122
##    temp_6_hrs temp_12_hrs temp_18_hrs temp_24_hrs dead moribund live avg_temp
## 55          8           8           8           9    0        0    1        9
##    dead_and_moribund trial
## 55                 0     3
```

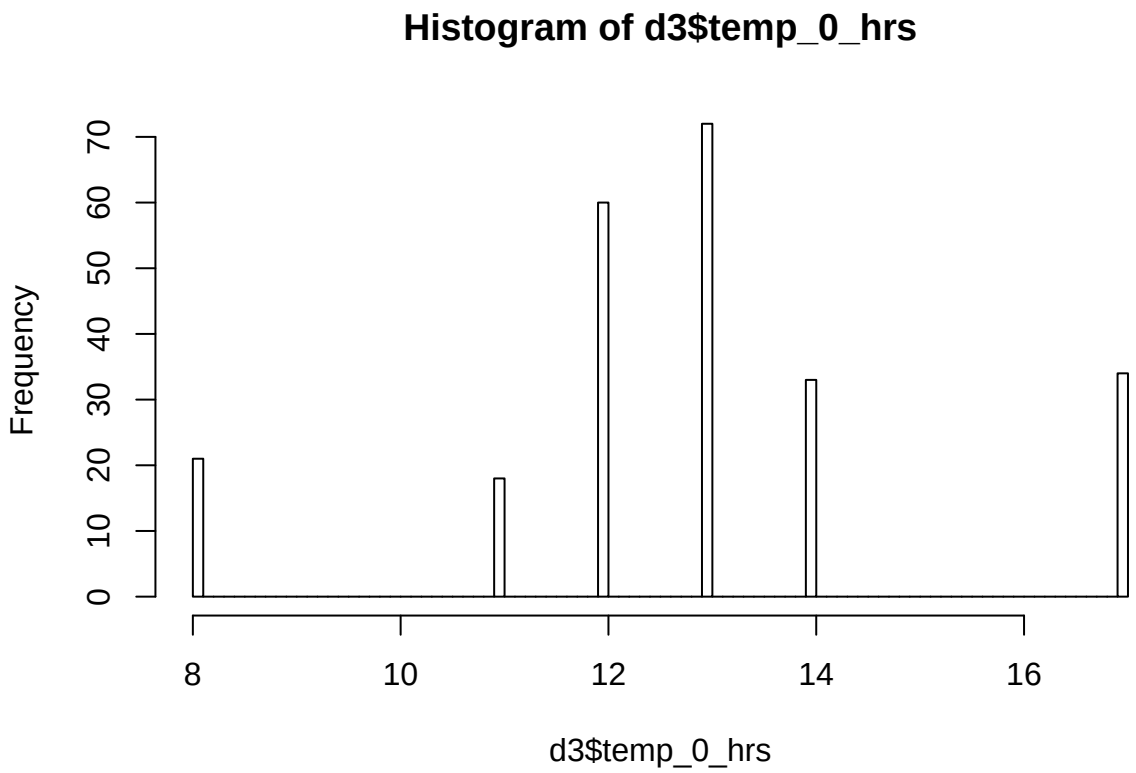
So we see there is only one observation above 100 degrees celcius, in the 55th row. It is 122 degrees. We can either disgard this observation if we do not feel it is reliable, or we can try and see if there was an obvious typo and thus obvious correction we can make to that entry. To inform our decision, we'll look at the other values from the same site/date combination.

```
# We use the '&' to specify two filtering criteria (location AND date)
# We could use '|' to specify rows which fit ONE OF those criteria (location OR date)
d3[d3$location=="Burdwoods" & d3$date=="June 20 2012",]$temp_0_hrs
```

```
##  [1]  12  12  12  12  12  12  12  12  12  12  12  12  12  12  12 122  12  12  12
## [20]  12  12  12  12  12  12  12  12  12  12  12  12  12  12  12  12  12
```

It is now pretty obvious that all other starting temperatures were 12 degrees celcius and whoever entered the data likely just accidentally added an extra 2 at the end. Knowing that, we can feel quite comfortable making the correction rather than discarding the data.

```r
d3$temp_0_hrs[55] <- 12
hist(d3$temp_0_hrs, breaks=100)
```



Sweet! There are other steps and checks we could perform, but now you have had a taste of the different kinds of data cleaning you'll likely encounter when preparing data for analysis. The final step at the end of data cleaning is to write your NEW dataframe to a file. Always write it to a new file so that you still have a copy of the original data, mistakes and all. This will ensure you can always go back, re-do steps, and double check your work.

```r
# We want to write our data to a comma separated variables format so we'll use write.csv()
write.csv(d3, "brand_new_clean_data.csv")
```

Other data cleaning/manipulation points

We covered a number of common cleaning activities here, but this is by no means an exhaustive list. You'll come across many other fun tasks you'll need to do. Other common tasks that we didn't have time to cover here but recommend you investing some time to learn (via online tutorials or some such thing) are:

1. Re-formatting dataframes (i.e. moving from long to wide data and vice versa)
2. Using and editing row names (we've only looked at column names)
3. Dealing with missing data (this could be a whole workshop series itself!)

While we didn't do any examples of this kind today, in our final 'putting it all together' session, we'll be combining our tasks from last week and this week to use `for loops` and `functions` to efficiently clean data.

Recap of Week 3:

This week we learned:

1. Why we bother cleaning data
2. How to avoid cleaning & general best practices
3. How to detect, locate, and correct errors in data (i.e. reassigning values)
4. How to subset data
5. How to plot to look for data outliers
6. We got a taster of the `TidyR` vs. `BaseR` debate

A further treatment of `BaseR` vs. `TidyR`

Packages are almost essential in the R environment, we wouldn't be able to do the vast majority of what we do nearly as easily without them. They are, some would argue, the whole reason R exists in the first place. Having said that, there is a school of thought that it is unimportant to understand the programming mechanisms that underpin many of the packages and functions we take for granted. We both vehemently disagree with this philosophy, and instead believe that without a good (or at least workable) understanding of what's going on 'under the hood', it will be very hard for users of R to fully utilize the software's capabilities. While both of us personally know stellar scientists and stellar biologists specifically that only use TidyR, we think that without an understanding of what's going on again 'under the hood', it will be difficult for anyone to a) move to a different language and be able to perform the same programming operations related to data management and cleaning without having to essentially re-learn all the fundamentals, b) be able to troubleshoot issues that come up when TidyR functions don't quite achieve the task at hand, or c) effectively understand the structures they are creating and altering when performing these operations.

Additionally, I would argue that the use of tidyR functions alongside baseR functions, which is often necessary, is incredibly difficult given that the whole purpose of TidyR is that it is written with a completely different syntactical expression and it is often impossible to use either TidyR functions in BaseR functions, or vice versa. Without droning too much on this topic, I encourage you to (if you are interested) look at some of the MANY opinions that have been shared online regarding this topic. Specifically, I point you to, for the sake of equal representation, one excellent essay written from a proponent of BaseR that essentially summarizes my personal thoughts on the topic (https://github.com/matloff/TidyverseSkeptic), and one from a TidyR instructor on the benefits of the Tidyverse (https://www.r-bloggers.com/why-learn-the-tidyverse/). One point that I will highlight, that I think is relevant, is that the tidyverse was built and is currently maintained by RStudio which is a for-profit company. While at first this seems normal, it is important to remember that this is not, I repeat NOT, how the open source software community usually works. Open source software, almost by definition, is typically built by people in their spare time, often not being paid directly for their work. This is not to say that those who work for RStudio or related companies are evil by any means, in fact I quite enjoy the products they produce! It is to say that the vast majority of literature that you will find on data management and data manipulation in the last 10 years has been written for the Tidyverse. This is almost purely because there are a large number of full-time staff members whose job it is to produce educational content promoting RStudios Tidyverse. This is not the case for BaseR (as it is still open source created), and the infrastructure in place that RStudio has developed, has no equal comparison in the BaseR community.

Both Emma and I agree that understanding how BaseR works is possibly the most important component of learning how to do programming in R. While it is true that other languages may be more suited for 'programming', as opposed to data manipulation and statistics (which are R's bread and butter), it is also true that biologists generally work in R, and the vast majority of statistics in biology are done in R. We are here to teach you R. However, you might go on to work another programming languages and/or continue on to perform complex analysis in R that cannot be done using the Tidyverse. In reality, none of us will probably be using R in another 20 to 30 years, another platform or language will have taken it's place, just as R has taken the place of Perl and S in terms of programming, and has somewhat/is still taking the place of the statistical softwares SAS and SPSS. As such, we think it's important to teach the basics and the foundations that, at least to some degree, hold true across the majority of high-level programming languages

on the market today. Thank you for reading this somewhat meandering and only mildly coherent opinion on the pros and cons of different methods of learning data manipulation. If you are interested in polarizing and often semantic debates of this kind, I encourage you to check out the plethora of opinions that have been published online in both print and video form on this topic of BaseR vs. TidyR.

Using Packages

As a topic distinctly different than that of TidyR and BaseR for data manipulation/management, I want to devote some space to packages broadly, particularly as they relate to packages for performing statistics. Packages are a critical part of the R environment; it's what makes R so user-friendly (compared to other programming languages), and allows people like us to analyze our data relatively easily. Packages are AMAZING for a ton of tasks, but there are some downfalls to them that are worth taking a few minutes to discuss.

The whole idea with packages is that there are operations (methods, functions, etc.) that we may want to do, often on data, that conceptually are fairly easy, but are more difficult, or lengthy from a programming perspective. Packages, built by developers and released open-source on CRAN, are essentially collections of functions that have been built to make our lives easier. In practice, what this means is that when we load a package in R and use one of that package's functions, we're actually running dozens and sometimes hundreds of lines of code that someone else has written, often in only one or two lines in our own scripts, allowing us to perform complicated programming operations very easily. Intuitively, this is a fantastic concept, and for our purposes as people who are not full-time computer scientists or software developers, it IS amazing. But in our opinion, we should be at least a little wary of packages, and we hope to convey why we think this over the course of this series. In a nutshell, while it makes our lives easier to abstract all that code away to a package, it also means that other people are making decisions about how the functions we're using are run, and often what assumptions we're okay with making. This becomes particularly important when using packages for running statistics, and it is every R-user's job to take the time to read up on how particular package functions are performing statistical operations that make assumptions not specified by the end-user.

Essentially, packages can be useful, but they can also be a crutch. They can allow us to perform programming tasks and data manipulation/analysis tasks without really knowing what's going on, which can quickly become dangerous. There is always a line to walk in terms of 'How much do I really need to know?' vs. 'How much can I leave to the package?' and we're not pretending to know the answer to the question of how to walk that line, it will depend on you and your situation the time. We are NOT advocating that you don't use packages, in fact, we'll teach you how to use some of the most common ones! What we're trying to convey is that we should never loose sight of the fact that when we're using packages we're essentially letting other people make decisions and assumptions for us. Most of the time, that's totally okay, but it's good to keep in mind, and avoid situations where it might be detrimental.