# Practical R for MBM

*Chuck Frost*

*6/30/2020*

## Practical R for MBM field data manipulation and analysis

The purpose of this ongoing short course is to familiarize MBM personnel with the concepts of tidy data, basic data manipulation, and basic programming techniques as implemented in R. It is expected that participants have a working understanding of R and RStudio. We will not go into depth on other types of programming, specifically statistical programming and computer programming. The topics covered here will foster appropriate treatment of MBM data throughout the data life cycle and lead to cleaner and more useful data in the future.

## Topics

1. R, RStudio, packages, help, intro to functions
2. Loading and saving, data types and structures
3. Tidy Data
4. Working with dates
5. Creating functions, conditional statements, looping
6. Visualization

### 1.0.1 What is R?

An open-source programming environment that serves as:

1. A giant calculator
2. An interface for data analysis and visualization
3. An interface for data manipulation and file handling
4. An interface for a simple and efficient programming language (S, that became R)

R isn't the "method." For example, "We used R to estimate population size." yuck

### 1.0.2 What is RStudio?

An integrated development environment (IDE) for R.

Provides tools, menus, help, and easy access to the best parts of R.

### 1.1.1 Packages

Base R automatically provides you with many common functions.

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

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

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

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

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

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

Other useful functions exist thanks to R users. Collections of related functions get wrapped into packages.

If you're looking for something NOT in base R, you need to install the related package.

The CRAN provides R community tested and approved packages. Check the packages tab in RStudio for some you already have installed but (probably) not loaded. Just click them to load. If you know what package you need, there are 2 easy ways to install it. The first is by clicking the install button in the packages tab. Search the CRAN for the package you want and install it. The second is:

```
install.packages("PackageName")
```

Don't forget to load the package after installation, either by checking the box under packages, or by calling:

```
library(PackageName)
```

Note that in install.packages() you are searching a string in quotes, "PackageName" while in library() you are calling a package object without quotes, PackageName.

### 1.1.2 Installing from GitHub

If a package isn't uploaded to CRAN, it can still be made available through GitHub. Packages installed from GitHub require an intermediate step:

```
install.packages("devtools")

library(devtools)

install_github("USFWS/AKaerial", ref = "development")

library(AKaerial)
```

This code does 4 things:
1. Install devtools package
2. Load devtools package

3. Call install_github (from devtools), look for the GitHub account "USFWS" and repository "AKaerial" and load the package in the "development" branch
4. Load AKaerial package

Installing and loading AKaerial may take a while since it includes dependencies. Dependencies are packages that are specified within another package that must be included for that package to operate appropriately. Akaerial "depends" on several other packages and will check if you have these installed.

### 1.1.3 Functions

Once you have a package loaded, you can access functions written and included in that package. Typing the name of a function will spit out the code that makes up the function. This can be messy for a long or complicated function. Typing ?FunctionName will access the help file for a given function.

```
AdjustCounts

?AdjustCounts
```

This can be extremely helpful (if the help file is!) when troubleshooting why your code isn't working as you want it to. Note that the top of the help file also tells you FunctionName {PackageName} in case you can't figure out where your function is coming from. This could be the case if you have dozens of packages loaded or have borrowed someone else's code to help run your analysis. Don't underestimate the power of internet searches to find if something already exists to help you do your thing!

We will talk about functions in more depth later, but for now, the general idea is that functions take arguments, run processes, and (usually) return products saved as R objects.

```
numbers = c(1,2,3)

avg = mean(numbers)

avg
```

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

In this example, numbers is initiated as a vector of 3 integers: 1,2,3. We then run the function mean with numbers as its argument, and save the result as the object avg.

It can be helpful to think of functions as recipes and arguments as ingredients.

```
Mix = function(what.to.mix){
  dough = sum(what.to.mix)
  return(dough)
  }

Bake = function(what.to.bake, time.to.bake){

  cookies = what.to.bake * time.to.bake

  return(cookies)

}
```

```
my.ingredients = c("sugar", "eggs", "butter", "flour")

my.dough = Mix(my.ingredients)

my.cookies = Bake(my.dough, 10)
```

### 2.1.0 Loading Files

There are many ways to read data into R. We will go over the most common way data is stored and used in R; as a data frame. A data frame is a 2-dimensional array (table) where each column represents a variable (category) and each row represents a unique observation. Let's start with creating your own data frame. This might be useful if you have a small data set.

```
small.data = data.frame(
  name = c("Laura", "Zak", "Chuck"),
  size = c("small", "medium", "large"),
  tenure = c(6, "drifter",9)
    )
```

We can now check the structure of the data frame. This is a great first QA/QC check! If we know that tenure should be numeric, the structure should confirm it.

```
str(small.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : Factor w/ 3 levels "Chuck","Laura",..: 2 3 1
##  $ size  : Factor w/ 3 levels "large","medium",..: 3 2 1
##  $ tenure: Factor w/ 3 levels "6","9","drifter": 1 3 2
```

Notice how all 3 variables are shown as Factor, which is incorrect. Factors, in general are tough to work with. If we try to change the value here (use $ to grab a column and brackets to select a row [ ], or use only brackets to select [row, column]), it throws an error.

```
small.data$tenure[2] = 3
```

```
## Warning in `[<-.factor`(`*tmp*`, 2, value = structure(c(1L, NA, 2L), .Label
## = c("6", : invalid factor level, NA generated
```

```
small.data[2,3] = 3
```

```
## Warning in `[<-.factor`(`*tmp*`, iseq, value = 3): invalid factor level, NA
## generated
```

```
levels(small.data$tenure)
```

```
## [1] "6"       "9"       "drifter"
```

Since R couldn't determine the structure of our tenure column, it defaulted to calling it a Factor, which can take any value. The unique set of values gets set as the levels of that factor. If you try to define a value outside of that set, you will get an error that results in NA. This is (almost) never what you want. So we can fix this in 2 ways:

First, we can use the "as." functions in base R to force R to evaluate something as a different data type. In this case, we can't just let R evaluate the factor as numeric since it will then take a numeric representation of that particular level of a factor in the set of all levels of that factor. What a mess. We have to first convert the factor to a character string using as.character, then to a numeric using as.numeric. This is called wrapping functions, when you pass as an argument to a new function the output of another function without saving the intermediate result. In the outer-most function call to as.numeric, anything that R can't represent as numeric will get NA. This is already messy and hard to trace changes to your data!

```
small.data$tenure=as.numeric(as.character(small.data$tenure))

str(small.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : Factor w/ 3 levels "Chuck","Laura",..: 2 3 1
##  $ size  : Factor w/ 3 levels "large","medium",..: 3 2 1
##  $ tenure: num  6 NA 9
```

Or, since we are scripting our analysis, we can change it in our original code chunk and just re-run. Score 1 for a nice scripted workflow... but imagine doing this through thousands or hundreds of thousands of rows of data.

```
small.data = data.frame(
  name = c("Laura", "Zak", "Chuck"),
  size = c("small", "medium", "large"),
  tenure = c(6,3,9)
    )
```

While we are at it, lets just purge the factors from our data entirely. It is easy to make something a factor later, if we ever see the need. To do this, we just pass the argument stringsAsFactors = FALSE to our data frame function. Note that we have to add a comma after the declaration of our final column tenure.

```
small.data = data.frame(
  name = c("Laura", "Zak", "Chuck"),
  size = c("small", "medium", "large"),
  tenure = c(6,3,9),
  stringsAsFactors = FALSE
    )

str(small.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ name  : chr  "Laura" "Zak" "Chuck"
##  $ size  : chr  "small" "medium" "large"
##  $ tenure: num  6 3 9
```

Now we are clean... maybe. Our structure says our name and size columns are chr (character) and our tenure correctly shows (num) numeric. Now we want add a new column (greatness) using the values in an existing column. We can both create and define the values of the new column in one step.

```
small.data$greatness = small.data$tenure^3

str(small.data)
```

```
## 'data.frame':    3 obs. of  4 variables:
##  $ name     : chr  "Laura" "Zak" "Chuck"
##  $ size     : chr  "small" "medium" "large"
##  $ tenure   : num  6 3 9
##  $ greatness: num  216 27 729
```

```
small.data
```

```
##    name   size tenure greatness
## 1 Laura  small      6       216
## 2   Zak medium      3        27
## 3 Chuck  large      9       729
```

Now we have our 4 columns and the structure even gives us what it can display of the values. Now what if we want to add another row of data? The function rbind (row bind) makes it easy. As long as our new row is in the same order and type as the columns of our data frame, we can simply call:

```
small.data = rbind(small.data, c("Hannah", "small", 1, 1))

str(small.data)
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ name     : chr  "Laura" "Zak" "Chuck" "Hannah"
##  $ size     : chr  "small" "medium" "large" "small"
##  $ tenure   : chr  "6" "3" "9" "1"
##  $ greatness: chr  "216" "27" "729" "1"
```

```
small.data
```

```
##     name   size tenure greatness
## 1  Laura  small      6       216
## 2    Zak medium      3        27
## 3  Chuck  large      9       729
## 4 Hannah  small      1         1
```

As you can imagine, the same thing can be done for columns using cbind. Let's add another one:

```
small.data = cbind(small.data, interests=c("Bedazzling", "K-Pop", "Couponing", "Oranges"))

small.data
```

```
##     name   size tenure greatness  interests
## 1  Laura  small      6       216 Bedazzling
## 2    Zak medium      3        27      K-Pop
## 3  Chuck  large      9       729  Couponing
## 4 Hannah  small      1         1    Oranges
```

### 2.2.0 Reading in a data file

If you have a data file that exists (hopefully) as .csv or .txt, base R has a function that will read your data into a data frame for you. We will demo this with a .csv of fake aerial survey data. Remember that R works best (and sometimes only!) with tidy data. Your file should at least be rectangular (no loose columns, no long or dangling rows) and preferably will have column headings, though those can also be defined later. The function read.table will be the most generic and robust method, but there is also read.csv that saves a little time since you are predefining the formatting.

Plain text format is the simplest way to store text. One step up are delimited files. The most common types of delimited files and tab- and comma-delimited. Comma-delimited files are commonly saved as .csv (comma separated values). R does not care how your files are delimited, you just have to know the delimiting character(s) and pass them as arguments in the function call.

```
my.data = read.table("SomeData.txt", header = TRUE, sep = " ")
```

This tells R to find the file SomeData.txt (it will default to your working directory, but you can path it anywhere). It will open it, read the header row into the data frame my.data, then scan the file for spaces (sep = " ") and add entries between spaces to the values in each row. What could possibly go wrong with space-delimiting? Or even comma-delimiting?

To illustrate further concepts below we will be using an artificial data set I modified for this demo. It is located on the GitHub repository for this short course. To read it in, we will use read.csv, which tells R to expect a comma-delimited file. Oh, and we should also get rid of those nasty factors for now.

```
aerial=read.csv(
  "https://raw.githubusercontent.com/cfrost3/MBM_R_Short_Course/master/BLSC_2018_RawObs_Fake.csv",
                header=TRUE,
                stringsAsFactors=FALSE)

str(aerial)
```

```
## 'data.frame':    624 obs. of  16 variables:
##  $ Year      : int  2018 2018 2018 2018 2018 2018 2018 2018 2018 2018 ...
##  $ Month     : chr  "6" "6" "6" "6" ...
##  $ Day       : int  15 15 15 15 15 15 15 15 15 15 ...
##  $ Seat      : chr  "LF" "LF" "LF" "LF" ...
##  $ Observer  : chr  "FAKE" "FAKE" "FAKE" "FAKE" ...
##  $ Strata    : chr  NA NA NA NA ...
##  $ Transect  : chr  "19" "19" "19" "19" ...
##  $ Segment   : logi  NA NA NA NA NA NA ...
##  $ Flight_Dir: chr  "88" "88" "89" "89" ...
##  $ Wind_Dir  : chr  "45" "45" "45" "45" ...
##  $ Wind_Vel  : chr  "24" "24" "24" "24" ...
##  $ Lat       : num  61.1 61.1 61.2 61.2 61.2 ...
##  $ Long      : num  -166 -165 -165 -165 -165 ...
##  $ Species   : chr  "START" "SCAU4" "LTDUF4" "BLSC4" ...
##  $ Num       : chr  "19" "1" "1" "1" ...
##  $ Obs_Type  : chr  "open" "single" "single" "pair" ...
```

Hey! We have real (fake) data! Anything stand out immediately in the structure of the file?

### 2.3.1 Data Types

R has 4 basic data types you will use (6 in total, actually, including raw and complex):
1. Integer (1, 2, 3, 4)
2. Numeric (2, 4, 2.4, 45.48)
3. Character ("cat", "DOG", "15d", "and so on")
4. Logical (TRUE, FALSE)

### 2.3.2 Data Structures

Depending on how you combine data types, you can end up with one of several data structures:
1. Vector - a collection of all one data type
2. Matrix - multidimensional collection of vectors
3. Factor - nomical set of unique values and a vector of integer indices
4. List - open format to contain data structure elements
5. Data frame - our most common, discussed above

There are many other data structures in R, including user-defined and package-defined structures. We will mostly stick to the common ones.

### 2.3.3 Useful Data Structure Functions

Base R comes with many quick ways to assess your data structures. We will go into more depth later, but for now, explore commands such as:

```
a = c(1,2,3,4,10.1)

length(a)  #how long is the vector?
```

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

```
mean(a)  #mean value
```

```
## [1] 4.02
```

```
var(a)  #variance
```

```
## [1] 12.802
```

```
typeof(a)  #what data type?  (double means floating decimal numeric)
```

```
## [1] "double"
```

```
is.na(a)  #any NA or missing values?
```

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

```r
summary(a)  #some basic summarizing statistics
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    2.00    3.00    4.02    4.00   10.10
```

```r
max(a)  #maximum value
```

```
## [1] 10.1
```

```r
min(a)  #minimum value
```

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

```r
dim(small.data)  #what are the dimensions of a structure?
```

```
## [1] 4 5
```

```r
class(small.data)  #what class of structure is it?
```

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

```r
names(small.data)  #what are the names of the objects that make up the structure?
```

```
## [1] "name"      "size"      "tenure"    "greatness" "interests"
```

```r
unique(small.data$size)  #what are all of the unique values in a range?
```

```
## [1] "small"  "medium" "large"
```

### 3.0.0 Tidy Data

Tidy data refers to a data set that satisfies 3 conditions:
1. Every column is 1 variable
2. Every row is 1 observation
3. Every cell is 1 value

In R, tidy data lends itself seamlessly to the vectorized functions we discussed above (and thousands of others). In addition to summarizing functions, such as length(), mean(), max(), min(), etc., plot functions in R work sometimes exclusively and other times much cleaner and more efficiently with tidy data.

It is important to note here that quality controlled data (typos removed, missing data correctly recorded, etc.) and tidy data are 2 separate and important characteristics.

Messy data (data that is not tidy) is often the result of database design by a data collector, rather than an analyst or data manager. This is not meant to place blame. A data collector has one primary purpose; translate the raw observations in the field into some sort of tabular archive. That archive will generally, then, take the form and function that was easiest and fastest for the collector to enter the data (a very reasonable result since it is often preferrable to get data collected and archived quickly). A data analyst similarly can have one primary purpose; to analyze data. While the analyst would also like to do things relatively as quickly as possible, messy data can make analysis lengthy, difficult, costly, and sometimes impossible. In MBM, for example, we have spent easily over 90% of our time cleaning data instead of on analysis (and the general consensus among similar analysts across all fields is 75-80% of time on cleaning).

### 3.1.0 Common Problems

There are 2 common problems with data that we run into frequently in MBM:
1. Using values as column names
2. Multiple variables in one column

### 3.1.1 Values as Column Names

Consider the following data frame:

```
messy1 = data.frame(Species = rep(c("COGO", "BOOW"), each = 3),
                    Box=c(1:6),
                    Visit1 = c(5,6,6,4,5,4),
                    Visit2 = c(5,6,5,4,5,7),
                    Visit3 = c(0,6,5,4,5,0),
                    Visit4 = c(NA,0,0,1,0,NA),
                    Visit5 = c(NA,NA,NA,0,NA,NA),
                    stringsAsFactors = FALSE
                    )


messy1
```

```
##   Species Box Visit1 Visit2 Visit3 Visit4 Visit5
## 1    COGO   1      5      5      0     NA     NA
## 2    COGO   2      6      6      6      0     NA
## 3    COGO   3      6      5      5      0     NA
## 4    BOOW   4      4      4      4      1      0
## 5    BOOW   5      5      5      5      0     NA
## 6    BOOW   6      4      7      0     NA     NA
```

Imagine they are visits to nests of 2 different species and the counts of the eggs that were found on each visit. Why is this messy? Looks clean enough. Easy enough for the data collector. Notice how columns 3-7 are labeled? This is a prime example of using a value for a variable (in this case, visit number) as the column header. It now becomes awkward for those nests that don't share the same number of visits. As an analyst, how can I script a quick summary of the number of times a nest was visited? Maybe by counting the number of cells that have a value greater than or equal to 0 in a row?

```
#give me the total of row 1 values greater than or equal to 0
v = sum(messy1[1,]>0)

v
```

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

Ouch. Oh yeah, the NA values. A sum containing an NA is always NA. So let's remove those.

```
#give me the total of row 1 values greater than or equal to 0, and skip the NAs
v = sum(messy1[1,]>0, na.rm = TRUE)

v
```

```
## [1] 4
```

Ok, looks like it worked. But not when we spot check it. We forgot that Box will (probably) always be greater than or equal to 0. So let's tack on a modifier.

```
#give the total of row 1 values >= to 0, skip the NAs, subtract 1 for Box
v = sum(messy1[1,]>0, na.rm = TRUE)-1

v
```

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

Well, we now correctly show 3 visits. But what could possibly go wrong with this in a larger script? Or imagine in an ACTUAL data set, where we have dozens of columns. This is the kind of thing that would drive an analyst bonkers AND would be a nightmare to fix after it has been done for years on a large data set. We won't even attempt to reshape this using base R commands. Lucky for us, we have the tidyverse package.

Tidyverse is actually a collection of packages developed by data scientists, for data scientists. When you install tidyverse, you get them all. It streamlines importing, cleaning, tidying, visualizing, analyzing, and reporting any size data set. Most (all) of the functions and functionality available in the tidyverse are also possible in base R, but often with significantly more (and generally more confusing) code. We can't possibly describe them all, but we will hit some of the main functions.

First, let's install the tidyverse package. This may take a while depending on how many of the packages you already have. We will also reload our fake aerial survey data to play with.

```
install.packages("tidyverse")
```

```
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 3.6.3

## -- Attaching packages ---------------------------------------------------------------------------

## v ggplot2 3.3.1     v purrr   0.3.3
## v tibble  3.0.1     v dplyr   1.0.0
## v tidyr   1.1.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0

## Warning: package 'ggplot2' was built under R version 3.6.3

## Warning: package 'tibble' was built under R version 3.6.3

## Warning: package 'tidyr' was built under R version 3.6.3

## Warning: package 'readr' was built under R version 3.6.1

## Warning: package 'purrr' was built under R version 3.6.3

## Warning: package 'dplyr' was built under R version 3.6.3
```

```
## Warning: package 'forcats' was built under R version 3.6.3
```

```
## -- Conflicts ---------------------------------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
aerial=read.csv(
  "https://raw.githubusercontent.com/cfrost3/MBM_R_Short_Course/master/BLSC_2018_RawObs_Fake.csv",
                header=TRUE,
                stringsAsFactors=FALSE)
```

If you recall, we used the $ operator to reference a column in a data frame. Tidyverse provides a cleaner syntax with the select function.

```
#grab Transect, Species, and Num only from aerial
aerial.subset = select(aerial, Transect, Species, Num)

head(aerial.subset)  #only show me the first few rows
```

```
##   Transect Species Num
## 1       19   START  19
## 2       19   SCAU4   1
## 3       19  LTDUF4   1
## 4       19   BLSC4   1
## 5       19   SCAU2   1
## 6       19   BLSC4   1
```

Quick, easy, and intuitive. Now let's only keep observations of 3 or more.

```
aerial.subset2 = filter(aerial.subset, Num > 2) #give me observations greater than 2

head(aerial.subset2)
```

```
##   Transect Species Num
## 1       15   SCAU1   5
## 2       16   BLSC3   3
## 3       27   START  27
## 4       27   ENDPT  27
## 5       26   START  26
## 6       26  BLSCF2   4
```

These are useful, but notice how we keep having to save out our results as new names with an increasing number on the end? This will get messy and out of control fast as we add on more filters and subsets and summaries. To combat this, the Tidyverse (specifically the magrittr package) has the pipe operator (%>%). The pipe operator chains together a series of functions and uses the results of the previous pipes in the subsequent pipes. It makes for a cleaner sequence that you can track later. For example, the sequence below will subset our columns and filter to bigger observations as we did above, but also filter out the START and ENDPT observations, then count the number of observations by species that met the conditions leading up to the aggregate.

```
obs.by.species = aerial %>%

  #grab Transect, Species, and Num only from aerial
  select(Transect, Species, Num)  %>%

  #Num greater than 2, but remove START and ENDPT observations
  filter(Num > 2 & !(Species %in% c("START", "ENDPT")) ) %>%


  aggregate(Num~Species, .,FUN=length)

  names(obs.by.species)[2]="N.obs"

obs.by.species
```

```
##   Species N.obs
## 1   BLSC3     2
## 2   BLSC4     1
## 3  BLSCF2     1
## 4  BLSCF3     2
## 5   LTDU3     1
## 6   SCAU1     6
## 7   SCAU2     2
## 8   SCAU3     7
```

We have a lot going on there. Note the command %in% that will only keep string values that it finds within a set of other string values. But in this case, I wrap it in !, which can be read as "not." So that line becomes "filter down to observations greater than 2 and where Species is not START or ENDPT."

Now that we have the pipe in our toolbox, let's revisit the actual messy data problem.

```
messy1
```

```
##   Species Box Visit1 Visit2 Visit3 Visit4 Visit5
## 1    COGO   1      5      5      0     NA     NA
## 2    COGO   2      6      6      6      0     NA
## 3    COGO   3      6      5      5      0     NA
## 4    BOOW   4      4      4      4      1      0
## 5    BOOW   5      5      5      5      0     NA
## 6    BOOW   6      4      7      0     NA     NA
```

We have 6 nest boxes that were visited up to 5 times and we just want a summary of the number of visits and the average number of eggs in the nest each visit. We have incorrectly recorded values of our variable Visit as column headers. Let's fix it.

```
tidy1 = messy1 %>%
  pivot_longer(c("Visit1", "Visit2","Visit3","Visit4","Visit5",),
               names_to = "Visit",
               values_to = "eggs")

tidy1
```

```
## # A tibble: 30 x 4
##    Species   Box Visit   eggs
##    <chr>   <int> <chr>  <dbl>
##  1 COGO        1 Visit1     5
##  2 COGO        1 Visit2     5
##  3 COGO        1 Visit3     0
##  4 COGO        1 Visit4    NA
##  5 COGO        1 Visit5    NA
##  6 COGO        2 Visit1     6
##  7 COGO        2 Visit2     6
##  8 COGO        2 Visit3     6
##  9 COGO        2 Visit4     0
## 10 COGO        2 Visit5    NA
## # ... with 20 more rows
```

This result is... ehhhh. It looks ok, but we can do better.

```
tidy1 = messy1 %>%
  pivot_longer(
   cols = starts_with("Visit"),
   names_to = "Visit",
   names_prefix = "Visit",
   values_to = "Eggs",
   values_drop_na = TRUE
 )

head(tidy1,10)
```

```
## # A tibble: 10 x 4
##    Species   Box Visit  Eggs
##    <chr>   <int> <chr> <dbl>
##  1 COGO        1 1         5
##  2 COGO        1 2         5
##  3 COGO        1 3         0
##  4 COGO        2 1         6
##  5 COGO        2 2         6
##  6 COGO        2 3         6
##  7 COGO        2 4         0
##  8 COGO        3 1         6
##  9 COGO        3 2         5
## 10 COGO        3 3         5
```

We took messy1 and applied pivot_longer, which will translate our column values into expanded row values. We further parameterized it by asking for all columns that started with Visit (cols = starts_with("Visit")), and removed the prefix Visit (names_prefix = "Visit") and sent the remainder to a new Visit column (names_to = "Visit"). We then took the row values (counts of eggs, though you can't tell from the data) and sent them to the appropriate new Species-Box-Visit row combination and labelled the new column Eggs (values_to = "Eggs"). Finally, we removed the NAs. These were causing problems above and since they were really just empty cells and not visits, they are better off gone. Now we can cleanly summarize our data!

```
vis = tidy1 %>%
  aggregate(Visit~Species + Box, ., FUN=max)
names(vis)[3] = "Num.Visits"
```

```
vis
```

```
##   Species Box Num.Visits
## 1    COGO   1          3
## 2    COGO   2          4
## 3    COGO   3          4
## 4    BOOW   4          5
## 5    BOOW   5          4
## 6    BOOW   6          3
```

```
avg.egg = tidy1 %>%
  filter(Visit == 1) %>%
  aggregate(Eggs~Species, ., FUN=mean)
names(avg.egg)[2] = "Avg.Eggs"

avg.egg
```

```
##   Species Avg.Eggs
## 1    BOOW 4.333333
## 2    COGO 5.666667
```

```
#Or using just tidyverse

tidy1 %>%
 filter(Visit == 1) %>%
 group_by(Species) %>%
 summarize(Avg.Eggs=mean(Eggs))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 2 x 2
##   Species Avg.Eggs
##   <chr>      <dbl>
## 1 BOOW        4.33
## 2 COGO        5.67
```

### 3.1.2 Multiple Variables in One Column

Another common problem in MBM data sets is combining the values of multiple variables into a single column. Sometimes this is what we really thought was the best way to store values, and other times we make excuses about the limitations of our data collection programs or devices to store things certain ways. The results are messy data either way, and this serves to increase the rate of decay of the actual information we can extract from our data. Consider the following (fake) data:

```
messy2 = data.frame(monthday=paste(rep(6,10), c(1:10), sep="/"), Grade=rep(c("A", "F"), 5))

messy2
```

```
##    monthday Grade
```

```
## 1       6/1     A
## 2       6/2     F
## 3       6/3     A
## 4       6/4     F
## 5       6/5     A
## 6       6/6     F
## 7       6/7     A
## 8       6/8     F
## 9       6/9     A
## 10     6/10     F
```

In this simple example, we have monthday, which is the month and day separated by a period, and a fictitious grade received on that day. We should really have month and day as their own columns. Let's separate.

```
messy2 %>%
    separate(monthday, into=c("Month", "Day"), sep="/")
```

```
##    Month Day Grade
## 1      6   1     A
## 2      6   2     F
## 3      6   3     A
## 4      6   4     F
## 5      6   5     A
## 6      6   6     F
## 7      6   7     A
## 8      6   8     F
## 9      6   9     A
## 10     6  10     F
```

That was suspiciously easy...so let's try it with our aerial data. The column Species sounds like it should be the species. But it isn't. It is actually 3 values: species, behavior, and distance. But that isn't the end of it. The behavior was only recorded if the bird was flying. Pretty messy. So let's tidy it.

```
unique(aerial$Species)
```

```
##  [1] "START"  "SCAU4"  "LTDUF4" "BLSC4"  "SCAU2"  "BLSCF3" "BLSC3"
##  [8] "BLSCF4" "SCAU3"  "BLSC2"  "LTDU4"  "SCAU1"  "LTDU3"  "BLSC1"
## [15] "ENDPT"  "LTDU2"  "LTDU1"  "BLSCF1" "BLSCF2" "WWSC2"
```

```
aerial.tidy = aerial %>%
 filter(!(Species %in% c("START", "ENDPT"))) %>%  #remove start and end points
 select(Lat, Long, Species, Num) %>% #filter to just 4 columns for illustration
 separate(Species, into=c("Species", "Distance"), sep = -1) %>% #take away the rightmost string value
 separate(Species, into=c("Species", "Behavior"), sep = 4)  #take away the first 4 string values


 head(aerial.tidy)
```

```
##       Lat      Long Species Behavior Distance Num
## 1 61.1457 -165.4586    SCAU                 4   1
## 2 61.1564 -164.8116    LTDU        F        4   1
```

```
## 3 61.1587 -164.6340    BLSC                4   1
## 4 61.1595 -164.5453    SCAU                2   1
## 5 61.1605 -164.4803    BLSC                4   1
## 6 61.1605 -164.4803    BLSC                4   1
```

```
unique(aerial.tidy$Species)
```

```
## [1] "SCAU" "LTDU" "BLSC" "WWSC"
```

An initial look at the Species column showed that the Distance was always the last value in the string. So we were able to tell separate that we need to pull that one out.

```
separate(Species, into=c("Species", "Distance"), sep = -1)
```

Then after pulling that one out, we saw that the species code was always the first 4 characters in the string, so we were able to tell separate to always give us those.

```
separate(Species, into=c("Species", "Behavior"), sep = 4)
```

With the pipe, we did it all in one command, and we can move on to bigger and better things, such as visualization!