Session 1: Introduction to R

Brian Lovett & Natalie Z. Kerr

10/18/2020

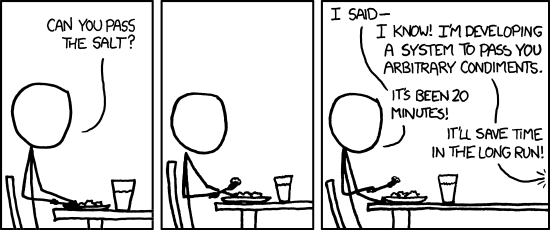
**Description:** This session will lay a coding foundation for R beginners. We will familiarize you with using the RStudio interface, RScripts and RMarkdown. Students will learn basic commands in the ever-changing R language, how to interpret errors and package installation. We will use these tools to wrangle and visualize data using Tidyverse packages.

## 1. Introduction to R Studio, R Scripts, RMarkdown/RNotebooks, and R Projects.

#### a. What is R?

R is a programming language. Essentially, R is a medium for dialogue between you and your computer about your data. We shall highlight the lingual aspects of R throughout this introduction.

#### b. Why use R?



xkcd comic: The General Problem

R is:

* Free
* Repeatable
* Easy to share
* Won’t alter your original data
* Saves time with reapplication
* Has community sourced packages
* Has a helpful online community

#### c. What does R do?

R is a way of telling your computer what you want done with your data. This language is geared toward mathematics, which is your computer’s forte. You will need to think like a computer. We will see that this language is very specific about what it wants and what it will give you- pay attention to these details!

#### d. What does R **not** do?

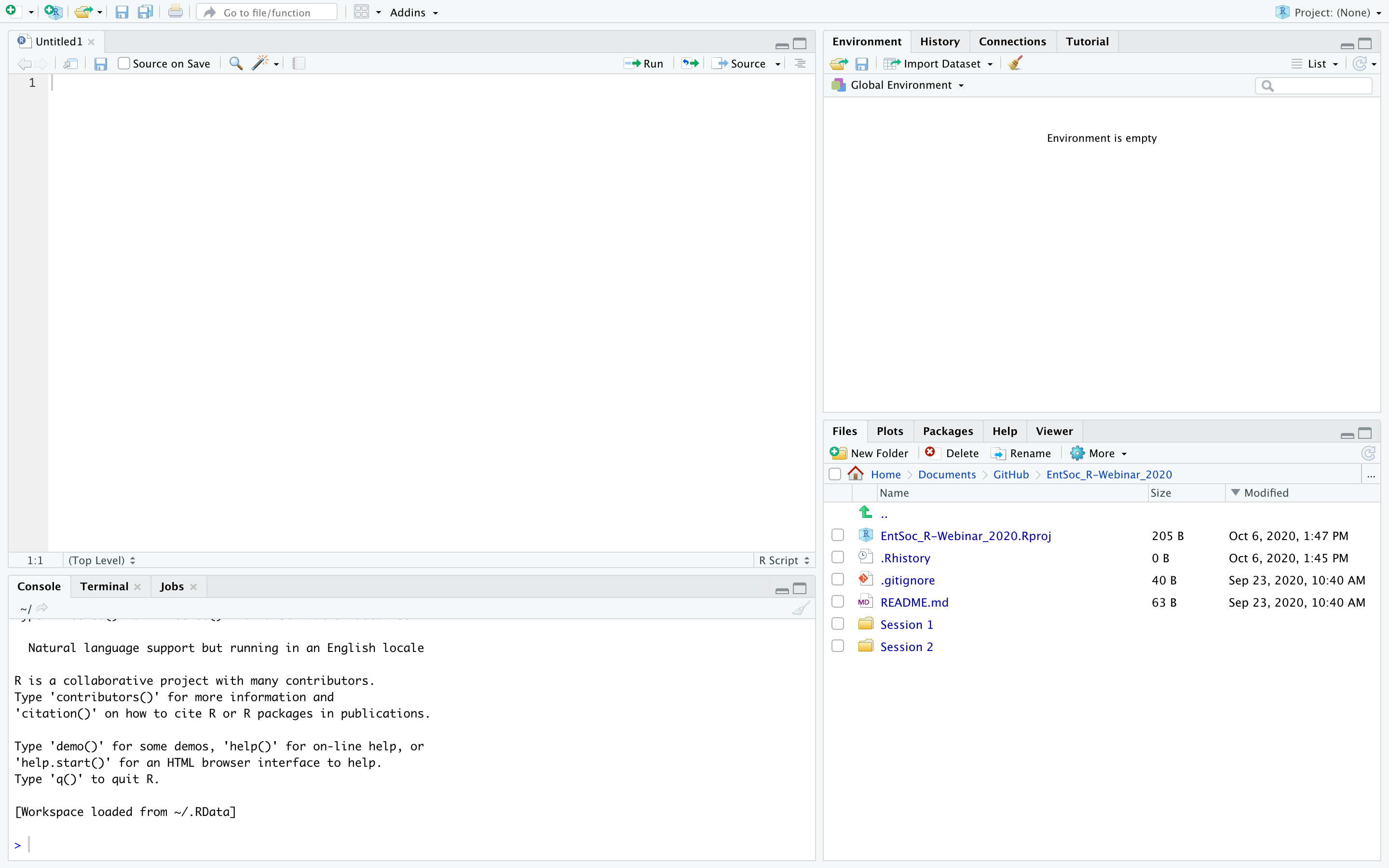
**Learning R is not learning statistics!** R can implement any number of statistical tests and analyses, and regardless of if they’re appropriate, it will run them on properly formatted data. Be careful to have a plan for what statistics you want to calculate \*\*\*\* you start coding.

#### e. Why do I need RStudio?

Primarily, RStudio provides a graphical user interface for coding in R. R alone is a console: a blank line asking for immediate instructions. Certainly, you can quickly code something like this, but part of coding is writing scripts that are reproducible. RStudio allows you to view:

1. Source (scripts and data)
2. Environment (saved objects and functions)
3. The console (where commands are run)
4. Files, Packages and Help (to orient you)

RStudio makes it easier to navigate between code, the console and output, while also displaying the characteristics of your R environment. You can think of RStudio as an “R organizer”.



The anatomy of the RStudio screen.

#### f. What are R scripts?

R scripts list commands you want to write, edit and save. In a simple example, a script may load data (command 1), calculate a statistic on that data (command 2), then plot that statistic (command 3). With this script in hand, you can instantly run the exact same analysis. You can also share the script and data and allow colleagues to replicate and contribute to analysis.

#### g. Why use RMarkdown or RNotebooks?

RMarkdown and and RNotebooks allow you to create sharable documents that combine a written narrative, code and results. RMarkdown syntax, which is used in RNotebooks, can be knit into pdfs or html files. (This was written via RMarkdown). RMarkdown and RNotebooks also allow you to run and view chunks of code within the document.

#### h. What are R Projects?

R Projects streamline your script development. A .Rproj file keeps track of what files you are viewing and will ensure you have a consistent working directory. R Projects open in their own R session, which allows you to jump right back into coding.

## 2. Taking action with functions (verbs)

#### a. What is a function?

Programming generally can be considered as a series of pipes: information goes in, the pipe does something (a function), then information comes out. Your aim is to ensure that appropriate (quality) information goes in, and that the “pipe” is doing the right thing. This pipe metaphor is appropriate too when you consider the idea of a “square peg in a round hole”. Half of the battle is understanding what type of information each function needs.

Functions can be thought of as verbs because they act on whatever objects you provide and often need additional information. Verbs not only tell you what is happening, but they also tell you something about when it happened and how many people were involved. The additional information you supply to functions are called arguments. Some are optional and others have defaults.

#### b. The anatomy of a function.

Regardless of what they do, functions share certain characteristics:

* Function name (can be any word)
* ( - marks the start of arguments
* Arguments
  + Each argument has a name and can be set to something
  + Some arguments will have defaults
  + Some arguments will be optional
  + Without names, order of arguments matters
* ) - marks the end of arguments

A function can be generalized as: function(argument, argument, argument) Let’s explore some functions:

#Note: Above boxes contain code that is sent to the console, and below boxes below are the output.  
  
#The function c (short for "combine") returns its contents as vector/list  
#The arguments for c are the objects you want to combine  
seuss <- c(1, "fish", 2, "fish")  
seuss

## [1] "1" "fish" "2" "fish"

#A colon is the coding equivalent of "through" 1:3 returns 1, 2 and 3. 3:1 returns 3, 2 and 1.  
nums <- c(1:100)  
nums

## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18  
## [19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36  
## [37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54  
## [55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72  
## [73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90  
## [91] 91 92 93 94 95 96 97 98 99 100

mean(x = nums)

## [1] 50.5

mean(nums)

## [1] 50.5

Since our object “seuss” contained numbers and words (words are “strings of characters” to your computer), this was formatted as a character class. A vector of 1-100 cooperates with the integer class, so the “nums” object is an integer class. This reliable behavior is built into the function c().

#### c. You can create your own functions!

You may want to write your own function to streamline your code or share specific workflows with others. If you find yourself copying and hard coding a few lines over and over, they may work better as a function. To write your own function, you need to specify the arguments of the function, the name of the function and the commands within the function.

#Here we will write a simple function that allows you to repeat "given numbers" (input: argument 1) x number of times" (n: argument 2) and take their sum. Within this custom function, we will use the functions rep.int(x, times) and sum(..., na.rm=F).  
  
repsum=function(input, n=2){  
 reps=rep.int(x=input, times=n)  
 sum(reps)  
}  
  
#Note: the object "reps" was created within our function and so will not be added to your environment  
#Also Note: we established a default for n if that argument is missing.  
  
repsum(input=nums)

## [1] 10100

#Relying on order for argument assignment  
repsum(nums, 3)

## [1] 15150

repsum(input=nums, n=4)

## [1] 20200

Functions work the same when nested, but they can be hard to read when you’re starting out.

repsum2=function(input, n=2){  
 sum(rep.int(x=input, times=n))  
}  
  
repsum2(nums)

## [1] 10100

repsum2(nums, n=4)

## [1] 20200

This nesting avoids creating the object “reps”, but it will be harder for future you and others to read.

#### e. R as a calculator

The R base language has intuitive math functions built in. These will follow order of operations.

1+2

## [1] 3

1+2\*3

## [1] 7

(1+2)\*3

## [1] 9

2^3

## [1] 8

3/4

## [1] 0.75

## 3. R as an object-oriented language (nouns)

Information is stored in R as objects, and R objects have a wide variety of classes. Some fundamental classes are:

* Number

as.numeric(x=5)

## [1] 5

numeric(length=5)

## [1] 0 0 0 0 0

* Character

as.character(5)

## [1] "5"

* Factor

factor(5)

## [1] 5  
## Levels: 5

* Integer

as.integer(5)

## [1] 5

* Matrix

matrix(5)

## [,1]  
## [1,] 5

* Data frame

data.frame(col=5)

## col  
## 1 5

* Logical

is.na(5)

## [1] FALSE

These examples are all ephemeral: R read our command and returned what we asked for, but objects are much more powerful when you store them. Typically you will store whatever objects you’re working with. You can store them in two ways:

five <- as.numeric(5)  
six = factor(6)  
  
#Running the name of an object will display that object  
five

## [1] 5

six

## [1] 6  
## Levels: 6

Since I deviously created these objects named “five” and “six” in different classes, let’s compare them. First, we can check the class for ourselves:

class(five)

## [1] "numeric"

class(six)

## [1] "factor"

five+five

## [1] 10

six+six

## Warning in Ops.factor(six, six): '+' not meaningful for factors

## [1] NA

We can also look at specific characteristics of these objects:

length(five)

## [1] 1

length(six)

## [1] 1

levels(six)

## [1] "6"

Levels are a powerful aspect of factors, let’s take a deeper look at this:

c(12:1)

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

c("apples", "oranges", "pears", "bananas")

## [1] "apples" "oranges" "pears" "bananas"

nums=factor(c(12:1))  
fruits=factor(c("apples", "oranges", "pears", "bananas"))  
  
nums

## [1] 12 11 10 9 8 7 6 5 4 3 2 1   
## Levels: 1 2 3 4 5 6 7 8 9 10 11 12

levels(nums)

## [1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12"

levels(fruits)

## [1] "apples" "bananas" "oranges" "pears"

as.character(nums)

## [1] "12" "11" "10" "9" "8" "7" "6" "5" "4" "3" "2" "1"

as.numeric(nums)

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

as.character(fruits)

## [1] "apples" "oranges" "pears" "bananas"

as.numeric(fruits)

## [1] 1 3 4 2

as.numeric(as.character(fruits))

## Warning: NAs introduced by coercion

## [1] NA NA NA NA

You can also reorder levels without reordering the factor itself.

new.order=c(1, 4, 3, 2)  
fruits

## [1] apples oranges pears bananas  
## Levels: apples bananas oranges pears

#To do this, we will use the brackets, which are an object element that allows you to specify order  
levels(fruits)

## [1] "apples" "bananas" "oranges" "pears"

levels(fruits)[new.order]

## [1] "apples" "pears" "oranges" "bananas"

fruits=factor(fruits, levels=levels(fruits)[new.order])  
fruits

## [1] apples oranges pears bananas  
## Levels: apples pears oranges bananas

The order of a factor can determine, for example, the order your data is displayed on a plot (as we shall see later).

#### 4. Creating loops and if/else statements

Loops allow you to iterate commands across a given list/vector (numeric or character). Similar to writing a function, these are initiated with a command, such as for(), while() and if(), followed by commands enclosed within curly brackets.

The arguments within the function for() are formatted like this: variable name “in” list (see the examples below). After the end of the curly brackets of the for() function, else() can be included with its own curly brackets.

while() loops and if()/else() statements allow you to program commands based on some condition (a logical expression). The argument for if() and while() is simply a conditional statement.

#A for() loop alone  
for(x in 1:10){  
 y=2\*x  
 print(c(x, y))  
 }

## [1] 1 2  
## [1] 2 4  
## [1] 3 6  
## [1] 4 8  
## [1] 5 10  
## [1] 6 12  
## [1] 7 14  
## [1] 8 16  
## [1] 9 18  
## [1] 10 20

#A for() loop with nested if()/else() statements  
#First we will set any variables needed for our loop  
VOWELS=c("A","E","I", "O", "U", "Y")  
alpha.dat=data.frame()  
  
for(x in LETTERS){  
 if(x %in% VOWELS){  
 alpha.dat=rbind(alpha.dat, (c(x, "is", "a", "vowel")))  
 }else{  
 alpha.dat=rbind(alpha.dat, c(x, "is", "a", "consonant"))  
 }  
}  
  
alpha.dat

## X.A. X.is. X.a. X.vowel.  
## 1 A is a vowel  
## 2 B is a consonant  
## 3 C is a consonant  
## 4 D is a consonant  
## 5 E is a vowel  
## 6 F is a consonant  
## 7 G is a consonant  
## 8 H is a consonant  
## 9 I is a vowel  
## 10 J is a consonant  
## 11 K is a consonant  
## 12 L is a consonant  
## 13 M is a consonant  
## 14 N is a consonant  
## 15 O is a vowel  
## 16 P is a consonant  
## 17 Q is a consonant  
## 18 R is a consonant  
## 19 S is a consonant  
## 20 T is a consonant  
## 21 U is a vowel  
## 22 V is a consonant  
## 23 W is a consonant  
## 24 X is a consonant  
## 25 Y is a vowel  
## 26 Z is a consonant

#A while() loop with nested if()/else() statements  
seed=10000  
counter=0  
  
#These sequences are central to the Collatz conjecture that posits they will always reach 1.  
while(seed>1){  
 counter=counter+1  
 if((seed %% 2)==0){  
 seed=seed/2  
 }else{  
 seed=3\*seed+1  
 }  
 print(c(seed, counter))  
}

## [1] 5000 1  
## [1] 2500 2  
## [1] 1250 3  
## [1] 625 4  
## [1] 1876 5  
## [1] 938 6  
## [1] 469 7  
## [1] 1408 8  
## [1] 704 9  
## [1] 352 10  
## [1] 176 11  
## [1] 88 12  
## [1] 44 13  
## [1] 22 14  
## [1] 11 15  
## [1] 34 16  
## [1] 17 17  
## [1] 52 18  
## [1] 26 19  
## [1] 13 20  
## [1] 40 21  
## [1] 20 22  
## [1] 10 23  
## [1] 5 24  
## [1] 16 25  
## [1] 8 26  
## [1] 4 27  
## [1] 2 28  
## [1] 1 29

Without print() in your loops, R will silently do what you ask and not return anything. You have to direct the results within your for loop to some object in order to keep them. Loops often function by creating intermediate variables that are overwritten as the sequence progresses, so only the final version of your object will enter your environment.

## 5. Packages, the tidyverse and tidy data

#### a. What is a package?

Everything we’ve discussed so far is part of the base R language. These are functions that R knows out of the box. Packages extend that language for specific purposes: it’s a vocabulary lesson or “extension pack” for R. Packages are a collection of functions for you to use, loosely organized by purpose or field. Some packages are designed for field-specific purposes and others are more general. Other scientists can upload their packages into repositories, which stores them and allows them to be easily downloaded. CRAN (The Comprehensive R Archive Network) is a major repository for R packages, but there are others, such as Bioconductor.

Packages will have manuals that explain each function in the package and each of its arguments. Packages will often also have tutorials or vignettes via vignette() that walk you through basic usage. These are a good jumping off point if you want to explore a new package. Packages and their documentation can be updated and maintained by scientists who write them.

To install a package, you can use the command install.packages(). You can install packages by supplying a character or vector of characters: “package” or c(“package 1”, “package 2”).

Use this command: >install.packages(“readr”)

Packages will be downloaded and stored as a binary that you can call with the command library(package). Even if you have installed a package, a new session of R will be unaware of it until you call it with library(). The argument for library is the name of the package, so it doesn’t have to be in quotes (a character to your computer).

library(readr)

The functions in the package readr are now available to our computer. We can check this by opening a help window on readr or clicking on the Packages tab. To access help on any function or package, type into the council ?package or ?function. You can also use the searchbar in the RStudio Help tab.

We can try: >?readr

#### b. What is the tidyverse?

The tidyverse is a collection of packages developed and maintained by data scientists at RStudio. These packages are ggplot2 (for data visualization), dplyr (for data manipulation), tidyr (to help tidy data), readr (to read data), purr (to streamline functions), tibble (for tidy data frames- tibble class), stringr (for wrangling character strings), forcats (for managing factors). Installing and loading “tidyverse” will install and load each of these packages. library(“tidyverse”) is the at the top of many R scripts.

Each of these packages has stellar documentation, and we will only touch on the wonderful things some of them can do. I highly recommend reading the documentation of tidyverse packages you think would be useful to you in full: <https://www.tidyverse.org/packages/>

#### c. What is tidy data?

The tidyverse is oriented around the concept of tidy data. This is data which is formatted such that each column represents a unique variable or measurement and each row represents a single experimental unit.

Let’s break down an example. If you’re conducting a simple experiment examining blood feeding preference for mosquitoes, you may design it so that you offer 20 female mosquitoes either of two sources of blood (horse or goose blood) and repeat this experiment in triplicate. You may measure how much time each female spends feeding.

To tidily format this data, you will give each individual mosquito her own row, and the columns will be: blood source (horse or goose), individual number (1-20), replicate (1-3) and time spent feeding(s). It may sound intuitive, but this basic principle for recording and thinking about your data allows for powerful, fast and (relatively) painless data processing.

This formatting will result in data that is easy for your computer to manage. #Show the data.

### 6. Working with data

Data analysis in R is streamlined by the various tidyverse packages. We will use some below.

R is certainly equipped to do any kind of data manipulation; however, particularly when you’re starting out, don’t be afraid to make a small change in some other program (like Excel) to speed up your analysis. As you learn R, even minor data manipulation will become easier in RStudio.

#### a. Loading data

There are a variety of ways to load data in R. The simplest is to code data in manually. A data frame specifies columns and its contents, so for the example above.

#Create each column  
blood=rep(c("goose", "horse"), each=60)  
individual=rep(c(1:20), times=6)  
replicate=rep(c(1:3), each=20, times=2)  
time=rnorm(n=120, mean=60, sd=30)  
  
#Combine into a data frame  
mosq.dat=data.frame(blood, individual, replicate, time)  
  
dim(mosq.dat)

## [1] 120 4

str(mosq.dat)

## 'data.frame': 120 obs. of 4 variables:  
## $ blood : chr "goose" "goose" "goose" "goose" ...  
## $ individual: int 1 2 3 4 5 6 7 8 9 10 ...  
## $ replicate : int 1 1 1 1 1 1 1 1 1 1 ...  
## $ time : num 76 35 37.9 19.8 105.5 ...

This would also work if our objects were nested within the data.frame() function. A tibble can be made the same way using the function tibble().

R also has data built in. You can peruse these datasets by running the function data(). These are immediately accessible to any R user simply by calling their names. Rather unfortunately, the only insect-oriented data is InsectSprays. This is described as “Effectiveness of Insect Sprays” and is described simply: “The counts of insects in agricultural experimental units treated with different insecticides.” Despite this simplistic view of IPM, which dates back to a data analysis paper from 1942, we will use this simple data for manipulation below.

dat=InsectSprays

Typically, you will import in files you have generated yourself. Base R has the function read.csv() and readr provides read\_csv(). The readr option will import data as a tibble and has more options that are intuitive. However, RStudio has an “Import Dataset” button, which gives you a user interface for either option. I recommend using this and checking the preview provided. Both will return in your console the command for importing that data.

We will import our own comma separated values file (csv) containing iNaturalist observations for Monarch butterflies from 2018-2020. This command was generated using the RStudio “Import Dataset” button, in which we specified the name iNat.dat (much easier to manage than “iNat\_monarch2018\_2020”)

iNat.dat <- read.csv("Data/iNat\_monarch2018\_2020.csv")

#### b. Manipulating data

Now that we have our data loaded, we can manipulate it. Let’s use some special characters to subsetting our InsectSpray data. $ will allow you to select specific columns. You can create new columns using this operator, as well.

dat

## count spray  
## 1 10 A  
## 2 7 A  
## 3 20 A  
## 4 14 A  
## 5 14 A  
## 6 12 A  
## 7 10 A  
## 8 23 A  
## 9 17 A  
## 10 20 A  
## 11 14 A  
## 12 13 A  
## 13 11 B  
## 14 17 B  
## 15 21 B  
## 16 11 B  
## 17 16 B  
## 18 14 B  
## 19 17 B  
## 20 17 B  
## 21 19 B  
## 22 21 B  
## 23 7 B  
## 24 13 B  
## 25 0 C  
## 26 1 C  
## 27 7 C  
## 28 2 C  
## 29 3 C  
## 30 1 C  
## 31 2 C  
## 32 1 C  
## 33 3 C  
## 34 0 C  
## 35 1 C  
## 36 4 C  
## 37 3 D  
## 38 5 D  
## 39 12 D  
## 40 6 D  
## 41 4 D  
## 42 3 D  
## 43 5 D  
## 44 5 D  
## 45 5 D  
## 46 5 D  
## 47 2 D  
## 48 4 D  
## 49 3 E  
## 50 5 E  
## 51 3 E  
## 52 5 E  
## 53 3 E  
## 54 6 E  
## 55 1 E  
## 56 1 E  
## 57 3 E  
## 58 2 E  
## 59 6 E  
## 60 4 E  
## 61 11 F  
## 62 9 F  
## 63 15 F  
## 64 22 F  
## 65 15 F  
## 66 16 F  
## 67 13 F  
## 68 10 F  
## 69 26 F  
## 70 26 F  
## 71 24 F  
## 72 13 F

dat$count

## [1] 10 7 20 14 14 12 10 23 17 20 14 13 11 17 21 11 16 14 17 17 19 21 7 13 0  
## [26] 1 7 2 3 1 2 1 3 0 1 4 3 5 12 6 4 3 5 5 5 5 2 4 3 5  
## [51] 3 5 3 6 1 1 3 2 6 4 11 9 15 22 15 16 13 10 26 26 24 13

#Here we are creating the column "n" with the value 30  
dat$n=30

This $ operator can also calculate between columns (across a row).

dat$dead=dat$n-dat$count

Double brackets function similar to the $ operator, except you can include a number or a name. Single brackets allow you to pick specific cells by coordinates: [row, column]. If a comma is not included, the value is assigned to column.

dat[[1]]

## [1] 10 7 20 14 14 12 10 23 17 20 14 13 11 17 21 11 16 14 17 17 19 21 7 13 0  
## [26] 1 7 2 3 1 2 1 3 0 1 4 3 5 12 6 4 3 5 5 5 5 2 4 3 5  
## [51] 3 5 3 6 1 1 3 2 6 4 11 9 15 22 15 16 13 10 26 26 24 13

dat[["count"]]

## [1] 10 7 20 14 14 12 10 23 17 20 14 13 11 17 21 11 16 14 17 17 19 21 7 13 0  
## [26] 1 7 2 3 1 2 1 3 0 1 4 3 5 12 6 4 3 5 5 5 5 2 4 3 5  
## [51] 3 5 3 6 1 1 3 2 6 4 11 9 15 22 15 16 13 10 26 26 24 13

#This command will extract the second row  
dat[2,]

## count spray n dead  
## 2 7 A 30 23

#This command will extract the second column  
dat[2]

## spray  
## 1 A  
## 2 A  
## 3 A  
## 4 A  
## 5 A  
## 6 A  
## 7 A  
## 8 A  
## 9 A  
## 10 A  
## 11 A  
## 12 A  
## 13 B  
## 14 B  
## 15 B  
## 16 B  
## 17 B  
## 18 B  
## 19 B  
## 20 B  
## 21 B  
## 22 B  
## 23 B  
## 24 B  
## 25 C  
## 26 C  
## 27 C  
## 28 C  
## 29 C  
## 30 C  
## 31 C  
## 32 C  
## 33 C  
## 34 C  
## 35 C  
## 36 C  
## 37 D  
## 38 D  
## 39 D  
## 40 D  
## 41 D  
## 42 D  
## 43 D  
## 44 D  
## 45 D  
## 46 D  
## 47 D  
## 48 D  
## 49 E  
## 50 E  
## 51 E  
## 52 E  
## 53 E  
## 54 E  
## 55 E  
## 56 E  
## 57 E  
## 58 E  
## 59 E  
## 60 E  
## 61 F  
## 62 F  
## 63 F  
## 64 F  
## 65 F  
## 66 F  
## 67 F  
## 68 F  
## 69 F  
## 70 F  
## 71 F  
## 72 F

#This command will extract the value from the first column and the second through fourth row  
dat[2:4,1]

## [1] 7 20 14

The subset() function can select rows from a data frame based on a logical expression (T/F statement). We can select rows where >6 died like this:

subset(x=dat, subset=dat$dead>6)

## count spray n dead  
## 1 10 A 30 20  
## 2 7 A 30 23  
## 3 20 A 30 10  
## 4 14 A 30 16  
## 5 14 A 30 16  
## 6 12 A 30 18  
## 7 10 A 30 20  
## 8 23 A 30 7  
## 9 17 A 30 13  
## 10 20 A 30 10  
## 11 14 A 30 16  
## 12 13 A 30 17  
## 13 11 B 30 19  
## 14 17 B 30 13  
## 15 21 B 30 9  
## 16 11 B 30 19  
## 17 16 B 30 14  
## 18 14 B 30 16  
## 19 17 B 30 13  
## 20 17 B 30 13  
## 21 19 B 30 11  
## 22 21 B 30 9  
## 23 7 B 30 23  
## 24 13 B 30 17  
## 25 0 C 30 30  
## 26 1 C 30 29  
## 27 7 C 30 23  
## 28 2 C 30 28  
## 29 3 C 30 27  
## 30 1 C 30 29  
## 31 2 C 30 28  
## 32 1 C 30 29  
## 33 3 C 30 27  
## 34 0 C 30 30  
## 35 1 C 30 29  
## 36 4 C 30 26  
## 37 3 D 30 27  
## 38 5 D 30 25  
## 39 12 D 30 18  
## 40 6 D 30 24  
## 41 4 D 30 26  
## 42 3 D 30 27  
## 43 5 D 30 25  
## 44 5 D 30 25  
## 45 5 D 30 25  
## 46 5 D 30 25  
## 47 2 D 30 28  
## 48 4 D 30 26  
## 49 3 E 30 27  
## 50 5 E 30 25  
## 51 3 E 30 27  
## 52 5 E 30 25  
## 53 3 E 30 27  
## 54 6 E 30 24  
## 55 1 E 30 29  
## 56 1 E 30 29  
## 57 3 E 30 27  
## 58 2 E 30 28  
## 59 6 E 30 24  
## 60 4 E 30 26  
## 61 11 F 30 19  
## 62 9 F 30 21  
## 63 15 F 30 15  
## 64 22 F 30 8  
## 65 15 F 30 15  
## 66 16 F 30 14  
## 67 13 F 30 17  
## 68 10 F 30 20  
## 72 13 F 30 17

Another useful function for subsetting is grep(). This function searches for patterns and returns where in a sequence that pattern occurs. We can use grep to find each row that contains our pattern “2” in the column count (i.e., 20, 12 and 2 would be included because they contain 2).

#grep() returns the row numbers  
grep("2", dat$count)

## [1] 3 6 8 10 15 22 28 31 39 47 58 64 69 70 71

#We can combine this with [] to subset those rows from our entire data frame  
#Note: we need to include a comma after grep() so that [] does not default to columns  
dat[grep("2", dat$count),]

## count spray n dead  
## 3 20 A 30 10  
## 6 12 A 30 18  
## 8 23 A 30 7  
## 10 20 A 30 10  
## 15 21 B 30 9  
## 22 21 B 30 9  
## 28 2 C 30 28  
## 31 2 C 30 28  
## 39 12 D 30 18  
## 47 2 D 30 28  
## 58 2 E 30 28  
## 64 22 F 30 8  
## 69 26 F 30 4  
## 70 26 F 30 4  
## 71 24 F 30 6

There are also dplyr options for subsetting that are very intuitive.

#Since this is our first time using dplyr, we will call the tidyverse which loads dplyr  
#install.packages('tidyverse')  
library(tidyverse)

## ── Attaching packages ──────────────────────────────────────────────────────── tidyverse 1.3.0 ──

## ✓ ggplot2 3.3.2 ✓ dplyr 1.0.1  
## ✓ tibble 3.0.3 ✓ stringr 1.4.0  
## ✓ tidyr 1.1.1 ✓ forcats 0.5.0  
## ✓ purrr 0.3.4

## ── Conflicts ─────────────────────────────────────────────────────────── tidyverse\_conflicts() ──  
## x dplyr::filter() masks stats::filter()  
## x dplyr::lag() masks stats::lag()

filter(dat, dat$count>6)

## count spray n dead  
## 1 10 A 30 20  
## 2 7 A 30 23  
## 3 20 A 30 10  
## 4 14 A 30 16  
## 5 14 A 30 16  
## 6 12 A 30 18  
## 7 10 A 30 20  
## 8 23 A 30 7  
## 9 17 A 30 13  
## 10 20 A 30 10  
## 11 14 A 30 16  
## 12 13 A 30 17  
## 13 11 B 30 19  
## 14 17 B 30 13  
## 15 21 B 30 9  
## 16 11 B 30 19  
## 17 16 B 30 14  
## 18 14 B 30 16  
## 19 17 B 30 13  
## 20 17 B 30 13  
## 21 19 B 30 11  
## 22 21 B 30 9  
## 23 7 B 30 23  
## 24 13 B 30 17  
## 25 7 C 30 23  
## 26 12 D 30 18  
## 27 11 F 30 19  
## 28 9 F 30 21  
## 29 15 F 30 15  
## 30 22 F 30 8  
## 31 15 F 30 15  
## 32 16 F 30 14  
## 33 13 F 30 17  
## 34 10 F 30 20  
## 35 26 F 30 4  
## 36 26 F 30 4  
## 37 24 F 30 6  
## 38 13 F 30 17

slice\_head(dat, prop=.25)

## count spray n dead  
## 1 10 A 30 20  
## 2 7 A 30 23  
## 3 20 A 30 10  
## 4 14 A 30 16  
## 5 14 A 30 16  
## 6 12 A 30 18  
## 7 10 A 30 20  
## 8 23 A 30 7  
## 9 17 A 30 13  
## 10 20 A 30 10  
## 11 14 A 30 16  
## 12 13 A 30 17  
## 13 11 B 30 19  
## 14 17 B 30 13  
## 15 21 B 30 9  
## 16 11 B 30 19  
## 17 16 B 30 14  
## 18 14 B 30 16

select(dat, c(count, spray))

## count spray  
## 1 10 A  
## 2 7 A  
## 3 20 A  
## 4 14 A  
## 5 14 A  
## 6 12 A  
## 7 10 A  
## 8 23 A  
## 9 17 A  
## 10 20 A  
## 11 14 A  
## 12 13 A  
## 13 11 B  
## 14 17 B  
## 15 21 B  
## 16 11 B  
## 17 16 B  
## 18 14 B  
## 19 17 B  
## 20 17 B  
## 21 19 B  
## 22 21 B  
## 23 7 B  
## 24 13 B  
## 25 0 C  
## 26 1 C  
## 27 7 C  
## 28 2 C  
## 29 3 C  
## 30 1 C  
## 31 2 C  
## 32 1 C  
## 33 3 C  
## 34 0 C  
## 35 1 C  
## 36 4 C  
## 37 3 D  
## 38 5 D  
## 39 12 D  
## 40 6 D  
## 41 4 D  
## 42 3 D  
## 43 5 D  
## 44 5 D  
## 45 5 D  
## 46 5 D  
## 47 2 D  
## 48 4 D  
## 49 3 E  
## 50 5 E  
## 51 3 E  
## 52 5 E  
## 53 3 E  
## 54 6 E  
## 55 1 E  
## 56 1 E  
## 57 3 E  
## 58 2 E  
## 59 6 E  
## 60 4 E  
## 61 11 F  
## 62 9 F  
## 63 15 F  
## 64 22 F  
## 65 15 F  
## 66 16 F  
## 67 13 F  
## 68 10 F  
## 69 26 F  
## 70 26 F  
## 71 24 F  
## 72 13 F

#### c. Transformation, grouping and calculations

Here we will further solidify the concept of pipes. Pipes in R look like this %>% and they originate from the package magrittr. RStudio recognizes pipes and will indent automatically when you return on a line ending in a pipe. Pipes send the output from one command into the next command. Pipes allow you to code complex things that are also readable: without pipes, you would need to create a number of intermediate objects or nest many functions to reach the same end.

In our first example, we will use pipes to group our observations by spray type, then take the mean number of insects remaining. We will use the following dplyr functions:

* select(“list of columns”)
* group\_by(“grouping list”)
* summarize(“column”=“calculation”)

#First we send our dat data frame into the pipe  
dat.stat= dat %>%  
#We've created new columns so we'll select our original two InsectSpray columns  
 select(count, spray) %>%  
#In the line below, we are grouping the data frame by   
 group\_by(spray) %>%  
#Once grouped, we will calculate the mean and stdev using the functions mean() and sd()  
 summarize(mn=mean(count), sd=sd(count))

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

This quickly tells us the very simple story that, whatever these sprays were in 1942, far fewer insects survived in sprays C, D and E.

Our last example with this data set is pivoting. pivot\_wide will transform your data to maximize columns (think “less tidy”) and pivot\_longer will do the opposite. Certainly you will work with data where one of these transformations is necessary. Call the vignette vignette(“pivot”).

#First, we will add an individual column to our InsectSpray dataset  
dat = dat %>%  
 group\_by(spray) %>%  
 mutate(individual=c(1:12))  
  
wide.dat= dat %>%  
 select(count, spray, individual) %>%  
 pivot\_wider(names\_from=individual, values\_from=count)  
  
wide.dat

## # A tibble: 6 x 13  
## # Groups: spray [6]  
## spray `1` `2` `3` `4` `5` `6` `7` `8` `9` `10` `11` `12`  
## <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 A 10 7 20 14 14 12 10 23 17 20 14 13  
## 2 B 11 17 21 11 16 14 17 17 19 21 7 13  
## 3 C 0 1 7 2 3 1 2 1 3 0 1 4  
## 4 D 3 5 12 6 4 3 5 5 5 5 2 4  
## 5 E 3 5 3 5 3 6 1 1 3 2 6 4  
## 6 F 11 9 15 22 15 16 13 10 26 26 24 13

#We can transform this back with the complementary function pivot\_longer  
long.dat = wide.dat %>%  
#Below, we can use - to signal which columns should not be pivoted. This is shorthand for the argument "cols", which alternatively needs a list of columns to pivot.  
 pivot\_longer(-spray, names\_to="individual", values\_to="count") %>%  
 select(spray, count, individual)  
  
long.dat

## # A tibble: 72 x 3  
## # Groups: spray [6]  
## spray count individual  
## <fct> <dbl> <chr>   
## 1 A 10 1   
## 2 A 7 2   
## 3 A 20 3   
## 4 A 14 4   
## 5 A 14 5   
## 6 A 12 6   
## 7 A 10 7   
## 8 A 23 8   
## 9 A 17 9   
## 10 A 20 10   
## # … with 62 more rows

we move on, we will make some adjustments to our iNaturalist data. This process of tidying or transforming your data visualization is called wrangling or munging.

#### d. Exporting tables

Straightforward options such as read.csv() and read\_csv() allow you to export objects out of R when you specify the object to export and the name for the new file. The default is to write directly to your working directory, so be careful to not overwrite anything important!

#### 7. Plotting in R

R has a built in plot() function, which can be used to visualize some results. However, this function is rarely used to produce publication quality figures. We will explore the package ggplot2, a tidyverse package, which provides a modular method for quickly creating multiple informative visualizations.

The package ggplot2 uses “grammar of graphics” (hence, “gg”) principles to allow you to layer on aspects of your plot. Once you learn this grammar, this approach to visualizing data allows you to make readable and intuitive adjustments to your plot.

The function ggplot() should contain an argument to define your data and a special kind of argument to define aesthetics: aes(). The arguments for aes() will change depending on the kind of plot you want, but here you will define the basic structure of your plot. You can define your x-axis (argument x), your y-axis (argument y) and what variable will be used to determine color (e.g., for points) or fill (e.g., for bars).

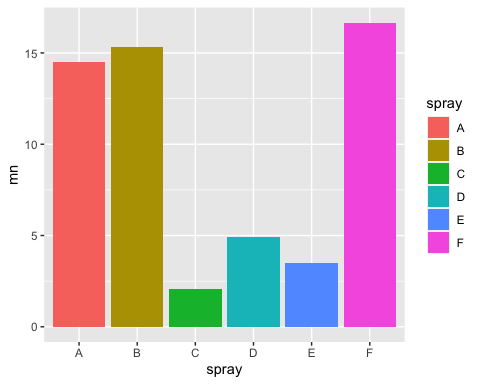
Once your ggplot() function is established, you will need to at least provide a plot type. These have very clear names, and can be accessed through any number of ggplot2 vignettes or data visualization libraries.

* geom\_bar() produces a bar plot
* geom\_point() produces a scatter plot
* geom\_line() produces a line plot

A basic ggplot bar plot will be defined as follows: ggplot(data=data, aes(x=column, y=column, color=column))+geom\_bar(stat=“identity”)

First, we will plot our InsectSpray data.

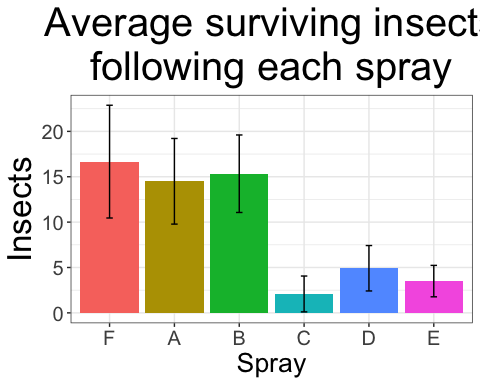
spray.plt=ggplot(data=dat.stat, aes(x=spray, y=mn, fill=spray))+geom\_bar(stat="identity")  
  
spray.plt

 This graph inherited a lot of aspects from ggplot. It is good for exploring data, but could use a little TLC sharing. To do this, we will use this current graph as a foundation and add on additional functions to alter the appearance. We can use the modularity of ggplot() to our advantage by creating some helpful objects first.

The object “theme” will contain minor tweaks to formatting, such as font size. Creating these in a separate object makes them easy to adjust and prevents them from cluttering up the command that makes our graph.

For similar reasons, we will create a separate object that contains the limits for our errorbar function.

theme = theme\_bw() + theme(text = element\_text(size=20), axis.title.x = element\_text(size=20), axis.text.x = element\_text(size=15), axis.text.y = element\_text(size=15), title = element\_text(size=25), legend.title = element\_text(size=15), legend.text = element\_text(size=20), plot.title = element\_text(hjust = 0.5))  
  
limits=aes(ymax=mn+sd, ymin=mn-sd)  
  
dat.stat$spray=factor(dat.stat$spray, levels=levels(dat.stat$spray)[c(6,1:5)])  
  
spray.dat2=ggplot(data=dat.stat, aes(x=spray, y=mn, fill=spray))+geom\_bar(stat="identity")+ geom\_errorbar(limits, width=0.1)+theme+ylab("Insects")+xlab("Spray")+ggtitle("Average surviving insects\nfollowing each spray")+ guides(fill=F)  
spray.dat2

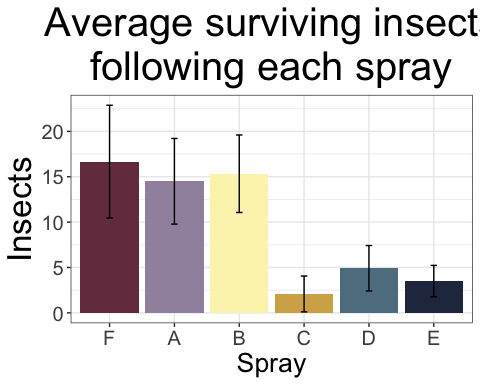


To take this one step further, we could determine our own bar colors. Typically, this process will be streamlined with a package containing available palettes to choose from. RColorBrewer is widely-used. We will install this and a younger package called MaizePal. MaizePal is not yet available on CRAN, so we will install it via github using a package called devtools.

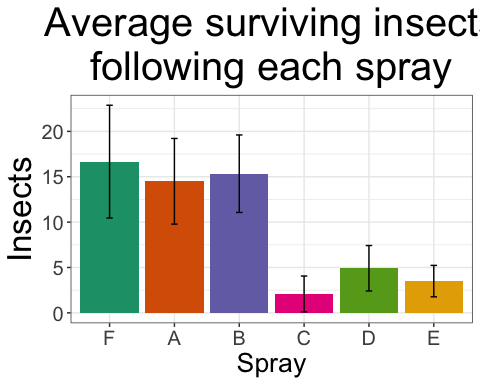
#We will install RColorBrewer from CRAN  
#install.packages('RColorBrewer')  
library(RColorBrewer)  
  
#We will install MaizePal using devtools  
#install.packages('devtools')  
#devtools::install\_github("AndiKur4/MaizePal")  
library(MaizePal)  
  
maize\_pal("HopiBlue")



palette1=maize\_pal("HopiBlue")  
  
spray.dat3 = spray.dat2 + scale\_fill\_manual(values=palette1)  
spray.dat4 = spray.dat2 + scale\_fill\_brewer(palette="Dark2")  
  
spray.dat3



spray.dat4



#### 8. Mapping in R

By popular demand, we will end with a primer on creating maps in R. I do not regularly map things for my work, but we can explore some broad principles and options to get you started.

First, a map to your computer is just a bunch of shapes with set positions. If you want to map

#install.packages(c('sf', 'maps', 'ggthemes', 'rnaturalearth', 'rnaturalearthdata', 'lubridate', 'gganimate', 'gifski'))  
library(sf)

## Linking to GEOS 3.8.1, GDAL 3.1.1, PROJ 6.3.1

library(ggthemes)  
library(rnaturalearth)  
library(rnaturalearthdata)  
library(rgeos)

## Loading required package: sp

## rgeos version: 0.5-5, (SVN revision 640)  
## GEOS runtime version: 3.8.1-CAPI-1.13.3   
## Linking to sp version: 1.4-2   
## Polygon checking: TRUE

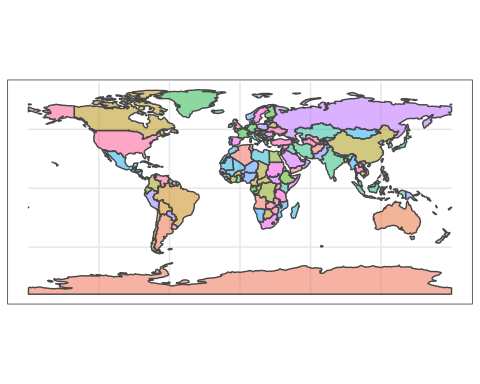
library(lubridate)

##   
## Attaching package: 'lubridate'

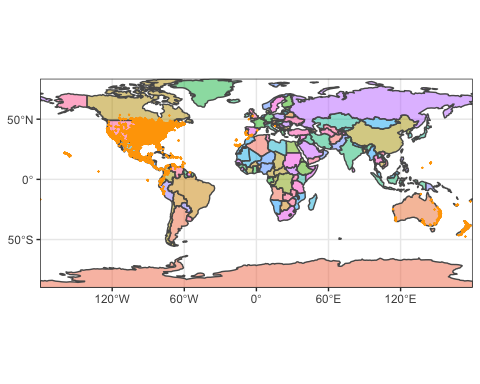
## The following objects are masked from 'package:rgeos':  
##   
## intersect, setdiff, union

## The following objects are masked from 'package:base':  
##   
## date, intersect, setdiff, union

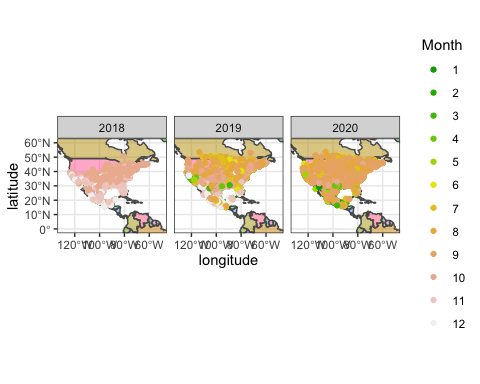
library(gganimate)  
library(gifski)  
  
world=ne\_countries(returnclass = 'sf')  
  
world.map <- ggplot(data=world) +  
 geom\_sf(aes(fill = name), alpha=0.5)+ guides(fill=F)+theme\_bw()  
  
world.map



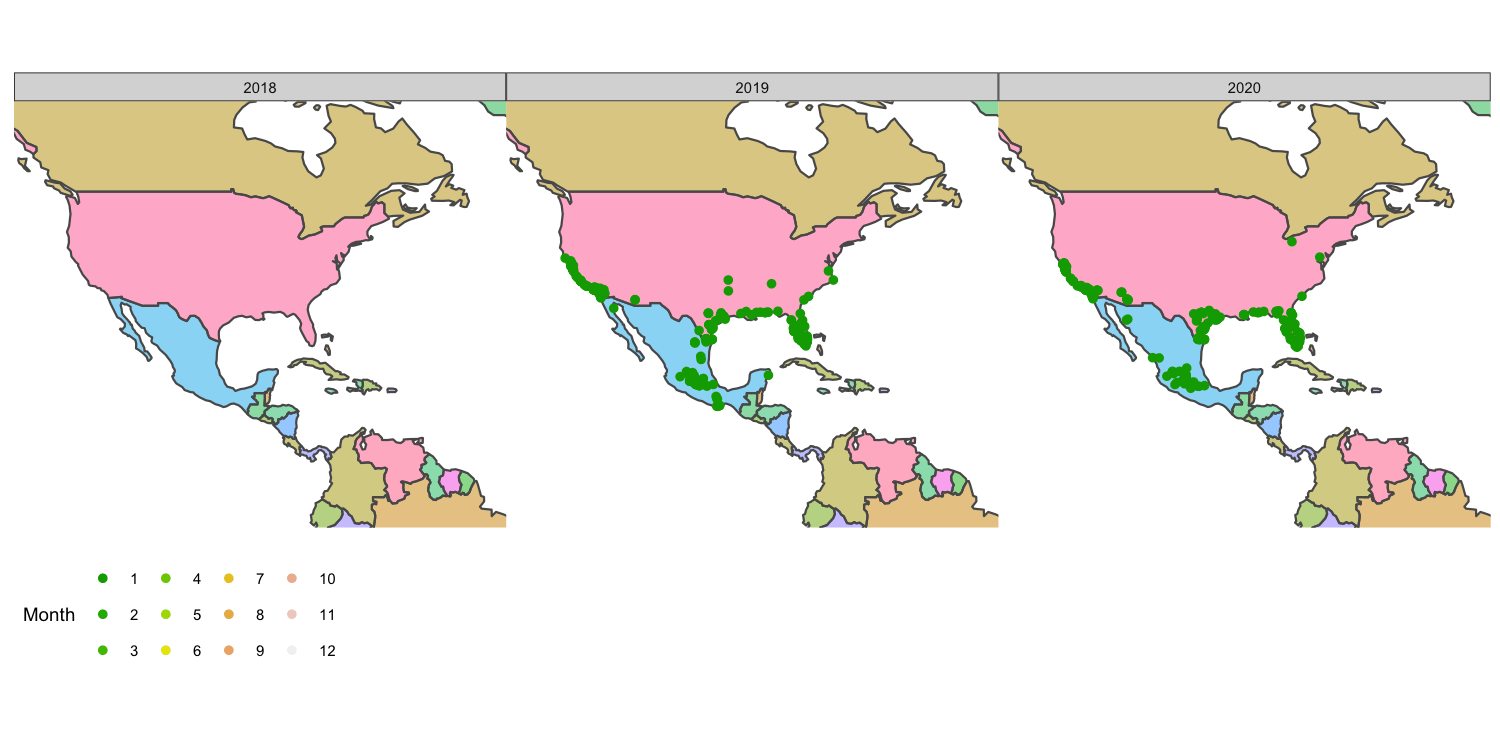
butterflies= iNat.dat %>%  
 filter(!is.na(latitude), !is.na(longitude), quality\_grade=="research", !is.na(positional\_accuracy)) %>%  
 select(latitude, longitude, observed\_on) %>%  
 st\_as\_sf(coords = c("longitude", "latitude"), crs = st\_crs(world))  
  
monarch.map= world.map + geom\_sf(data=butterflies, size=0.1, color="orange")+coord\_sf(expand=F)  
monarch.map



US.iNat.dat = iNat.dat %>%  
 filter(!is.na(latitude), !is.na(longitude), quality\_grade=="research", !is.na(positional\_accuracy), place\_country\_name %in% c("United States", "Canada", "Mexico")) %>%  
 mutate(Month=as.factor(month(x=observed\_on)), Year=year(x=observed\_on)) %>%  
 filter(!is.na(Year), !is.na(Month), longitude>-125)  
  
monarch.map <- world.map +  
 geom\_point(data=subset(US.iNat.dat), aes(x=longitude, y=latitude, color=Month)) +  
 coord\_sf(xlim = c(-130, -50), ylim = c(0, 60))+  
 scale\_color\_manual(values = terrain.colors(12))+facet\_wrap(~Year)  
  
monarch.map



monarch.map2 <- monarch.map +  
 transition\_states(Month, transition\_length = 1,  
 state\_length = 2, wrap=T) +  
 theme+theme\_map()+theme(legend.position = "bottom")

 Above is our gif that shows (month-by-month) where monarchs butterflies were observed on iNaturalist!

There is no one way to do something in programming! In this case, I supplied data that I manually pulled from iNaturalist. However, there is an R package that was written to streamline downloading and mapping iNaturalist data called “rinat”. Somewhat hilariously, their first example for usage is pulling monarch butterfly data too. Try it out!