Biology 305: Biostatistics

Dr. Jacob C. Cooper & Dr. Melissa Wuellner

Invalid Date

Table of contents

Preface

Welcome to Biology 105 at the University of Nebraska at Kearney! Material in this class was designed by Dr. Melissa Wuellner and adapted by Dr. Jacob C. Cooper for use in *R*.

In this class, you will learn:

1. The basics of study design, the importance of understanding your research situation before embarking on a full study, and practice creating research frameworks based on different scenarios.
2. The basics of data analysis, including understanding what kind of variables are being collected, why understanding variable types are important, and basic tests to understand univariate distributions.
3. Basic multivariate statistics, including ANOVA, correlation, and regression, for comparing multiple different groups.
4. The basics of coding and working in *R* for performing statistical analyses.

This site will help you navigate different homework assignments to perform the necessary *R* tests. Furthermore, this GitHub repository contains all of the homework dataframes, so you will *not* have to manually enter assignments if you use *R* to complete your assignments.

Welcome to class!

Dr. Jacob C. Cooper, BHS 321

1 Intro to *R*

In this class, we will be using *R* to perform statistical analyses. *R* is a free software program designed for use in a myriad of statistical and computational scenarios. It can handle extremely large datasets, can handle spatial data, and has wrappers for compatibility with *Python*, *Bash*, and other programs (even *Java*!).

2 Setup

First, we need to download *R* onto your machine. We are also going to download *RStudio* to assist with creating *R* scripts and documents.

2.1 Installing *R*

First, navigate to the *R* download and install page. Download the appropriate version for your operating system (Windows, Mac, or Linux). **Note** that coding will be formatted slightly different for Windows than for other operating systems.

Follow the installation steps for *R*, and verify that the installation was successful by searching for *R* on your machine. You should be presented with a coding window that looks like the following:

```
R version 4.4.1 (2024-06-14) -- "Race for Your Life"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20

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

  Natural language support but running in an English locale

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

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

>
```

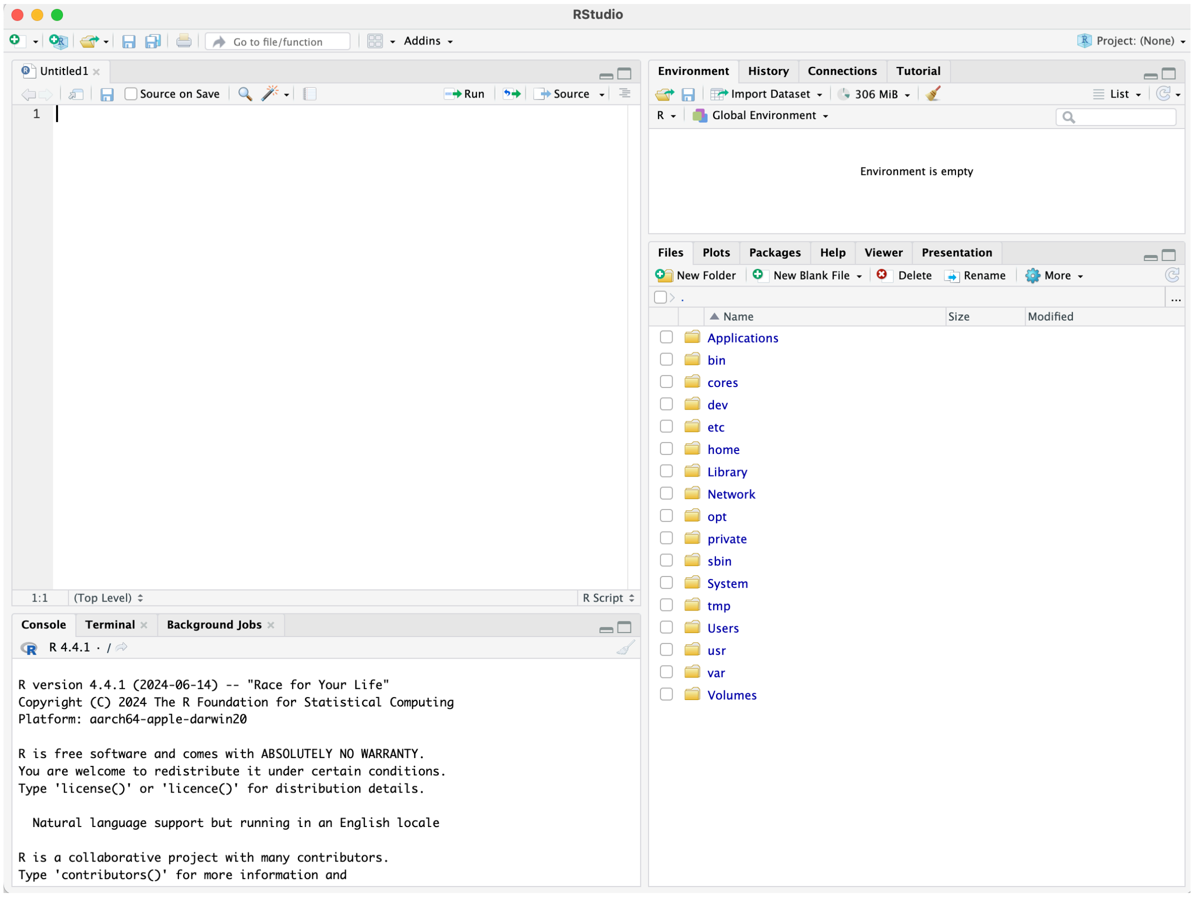
If that screen appears, congratulations! *R* is properly installed. If the install was not successful, please talk to Dr. Cooper and check with your classmates as well.

2.2 Installing *RStudio*

RStudio is a GUI (graphics user interface) that helps make *R* easier to use. Furthermore, it allows you to create documents in *R*, including websites (such as this one), PDFs, and even presentations. This can greatly streamline the research pipeline and help you publish your results and associated code in a quick and efficient fashion.

Head over the the *RStudio* download website and download "*RStudio* Desktop", which is free. Be sure to pick the correct version for your machine.

Open *RStudio* on your machine. You should be presented with something like the following:



```
R version 4.4.1 (2024-06-14) -- "Race for Your Life"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20

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

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
```

Figure 2.1: *RStudio* start window. Note that the screen is split into four different quadrants. Top left: *R* documents; bottom left: *R* program; top right: environment window; bottom right: plots, help, and directories.

In *RStudio*, the top left window is always going to be our coding window. This is where we will type all of our code and create our documents. In the bottom left we will see *R* executing the code. This will show what the computer is "thinking" and will help us spot any potential issues. The top right window is the "environment", which shows what variables and datasets are stored within the computers' memory. (It can also show some other things, but we aren't concerned with that at this point). The bottom right window is the "display" window. This is where plots and help windows will appear if they don't appear in the document (top left) window itself.

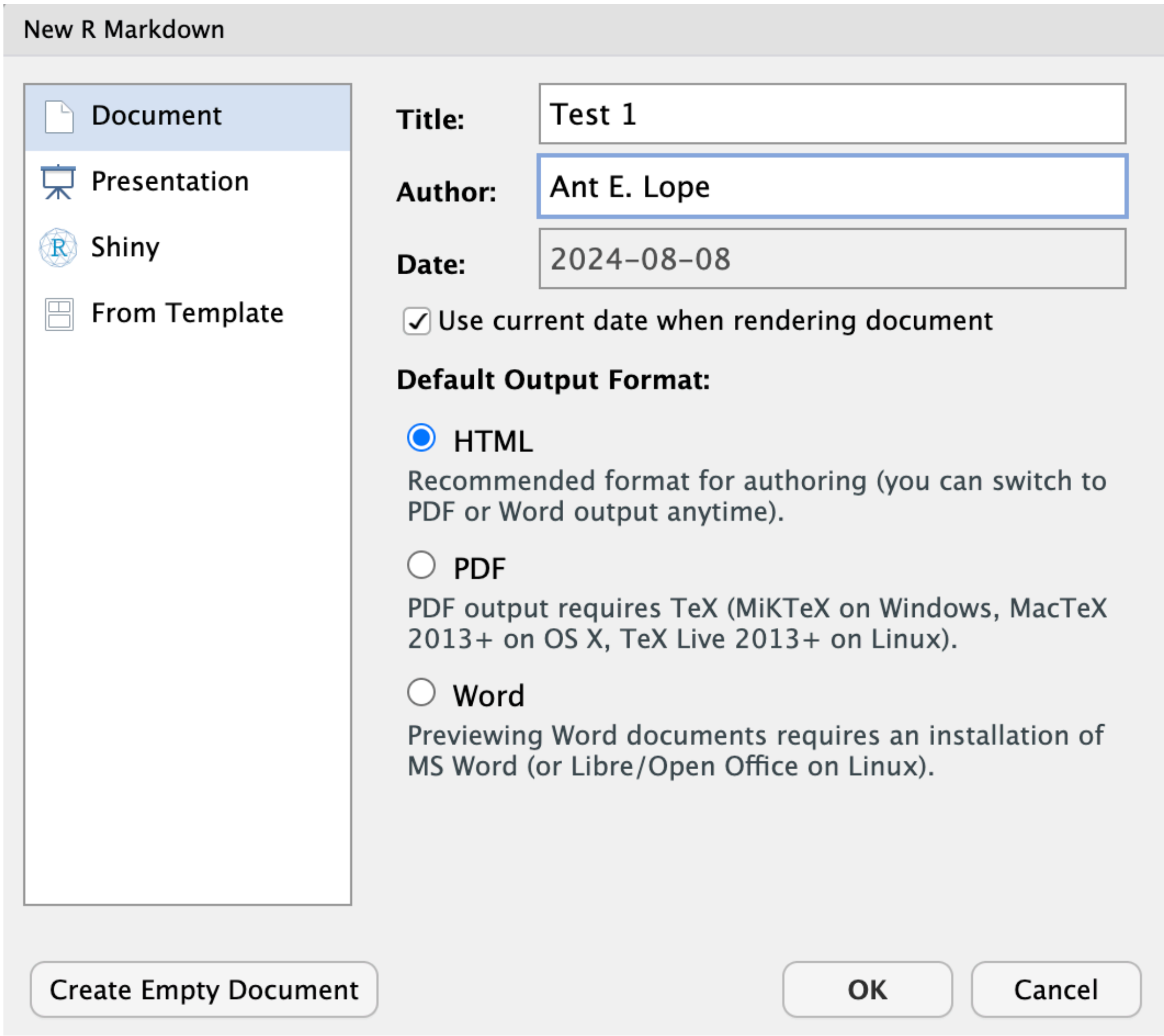
Now, we will create our first *R* document!

3 Creating an *RMarkdown* document

3.1 Setup

In this class, we will be creating assignments in what is called *RMarkdown*. This is a rich-text version of *R* that allows us to create documents with the code embedded. In *RStudio*, click the "+" button in the far top left to open the `New Document` menu. Scroll down this list and click on `R Markdown`.

A screen such as this will appear:



Title: Test 1

Author: Ant E. Lope

Date: 2024-08-08

☑ Use current date when rendering document

Default Output Format:

● HTML

Recommended format for authoring (you can switch to PDF or Word output anytime).

○ PDF

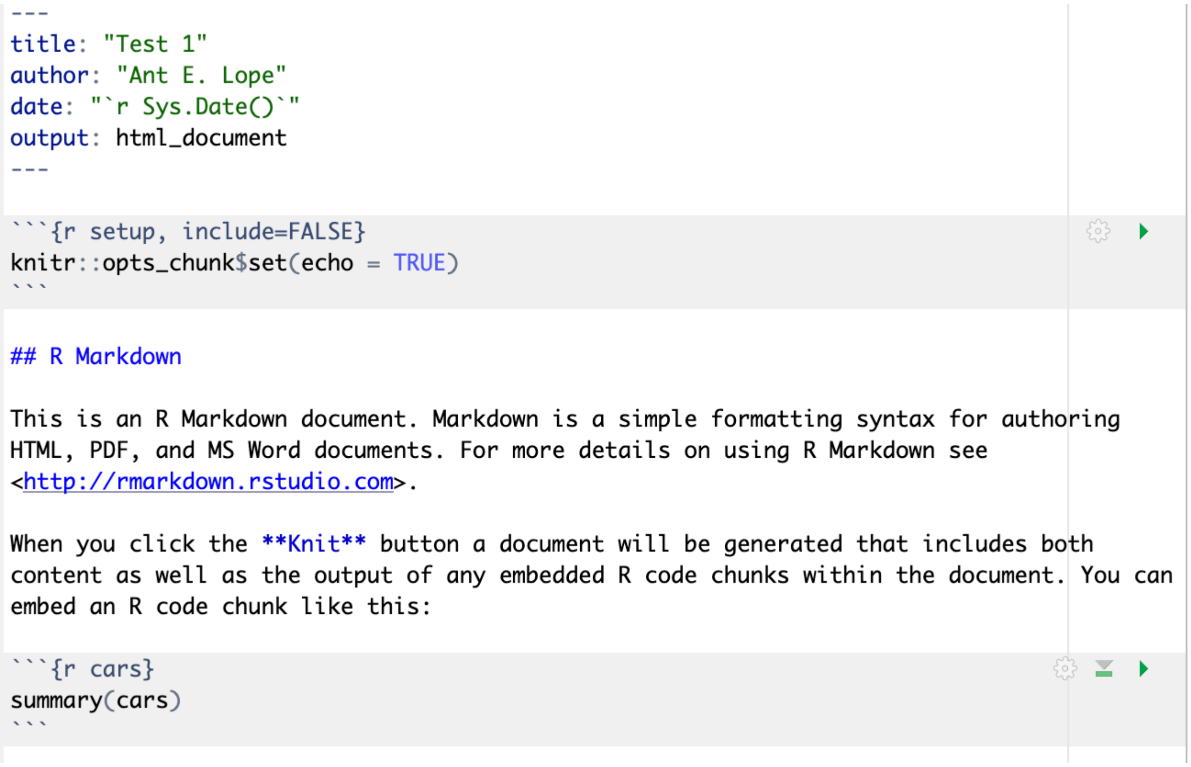
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

○ Word

Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

Figure 3.1: A new file window for an *RMarkdown* file.

After entering a title and your name and selecting `document` in the left hand menu, click `OK`.



~~~
---
title: "Test 1"
author: "Ant E. Lope"
date: "`r Sys.Date()`"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
~~~

R Markdown

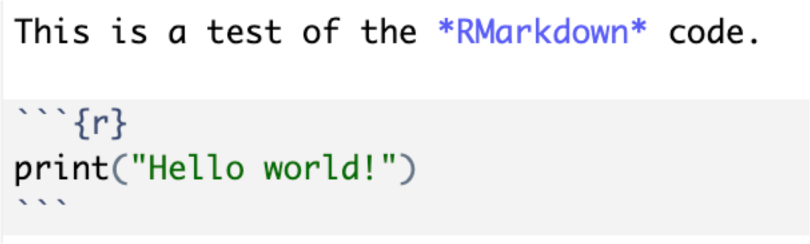
This is an R Markdown document. Markdown is a simple formatting syntax for authoring
HTML, PDF, and MS Word documents. For more details on using R Markdown see
<http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both
content as well as the output of any embedded R code chunks within the document. You can
embed an R code chunk like this:

```{r cars}
summary(cars)
```
```

Figure 3.2: An example of a markdown script.

In the image above, we can see what a "default" *RMarkdown* script looks like after creating the file. At the top of the document, between all of the dashes, we have the `yaml` header that tells *R* what kind of document will be created, who the author is, and tells it to use today's date. In this class, we will be saving documents as `html` as they are the easiest documents to create and save. These documents will include all of your code, text, and even any plots you may create!

Plain text in the document will be rendered as plain text in the document. (I.e., whatever you type normally will become "normal text" in the finished document). Lines preceded with `#` will become headers, with `##` being a second level header and `###` being a third level header, etc. Words can also be made italic by putting an asterisk on each side of the word (`*italic*`) and bold by putting two asterisks on each side (`**bold**`). URLs are also supported, with `<>` on each side of a URL making it clickable, and words being hyperlinked by typing `[words to show](target URL)`.

We also have code "chunks" that are shown above. A code chunk can be manually typed out or inserted by pressing `CTRL` + `ALT` + `I` (Windows, Linux) or `COMMAND` + `OPTION` + `I` (Mac). Everything inside a "code chunk" will be read as *R* code and executed as such. Note that you can have additional commands in the *R* chunks, but we won't cover that for now.
```

## 3.2 Using code chunks

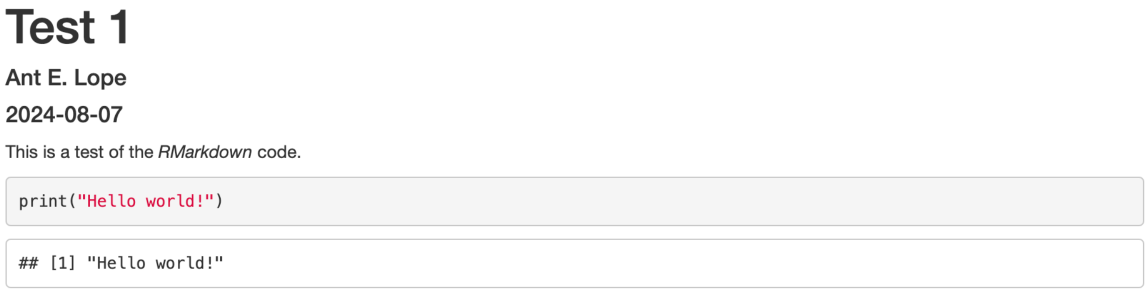
In your computer, erase all information except for the `yaml` header between the dashes on your computer. `Save` your file in a folder where you want your assignment to be located. It is important you do this step up front as the computer will sometimes save in random places if you don't specify a file location at the beginning. *Don't forget to save your work frequently*!



~~~
This is a test of the *RMarkdown* code.

```{r}
print("Hello world!")
```
~~~

Figure 3.3: Text to type in your *Rmarkdown* document.

After typing this into the document, hit `knit` near the top of the upper left window. *R* will now create an HTML document that should look like this:



Test 1

Ant E. Lope

2024-08-07

This is a test of the *RMarkdown* code.

```
print("Hello world!")
```

```
## [1] "Hello world!"
```

Figure 3.4: The output from the above code knitted into a document.

We can see now that the `HTML` document has the title of the document, the author's name, the date on which the code was run, and a greyed-out box with color coded *R* code followed by the output. Let's try something a little more complex. Create a new code chunk and type the following:

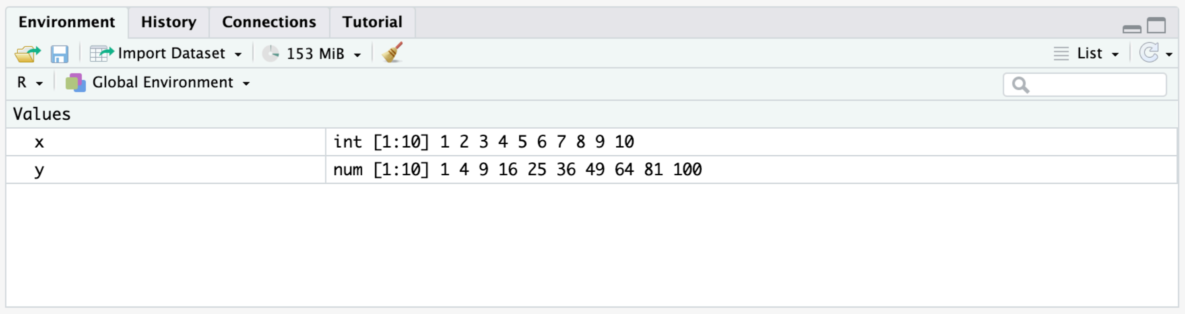
```r
x <- 1:10
```

This will create a variable in *R*, `x`, that is sequentially each whole number between 1 and 10. We can see this by highlighting or typing only the letter `x` and running that line of code by clicking `CTRL` + `ENTER` (Windows / Linux) or `COMMAND` + `ENTER` (Mac).

```
x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

If you look at the top right window, you will also see the value `x` in the environment defined as `int [1:10] 1 2 3 4 5 6 7 8 9 10`. This indicates that `x` is integer data spanning ten positions numbered 1 to 10. Since the vector is small, it displays every number in the sequence.



```
x    int [1:10] 1 2 3 4 5 6 7 8 9 10
y    num [1:10] 1 4 9 16 25 36 49 64 81 100
```

Figure 3.5: *RStudio* environment window showing saved objects. These are in the computer's memory.

Let's create another vector `y` that is the squared values of `x`, such that $y = x^2$. We can raise values to an exponent by using `^`.

```
y <- x^2
y
```

```
 [1]   1   4   9  16  25  36  49  64  81 100
```

Now we have the value `y` in the environment that is the square of the values of `x`. This is a `numeric` vector of 10 values numbered 1 to 10 where each value corresponds to a square of the `x` value. We can raise things to any value however, including x^x!

```
x^x
```

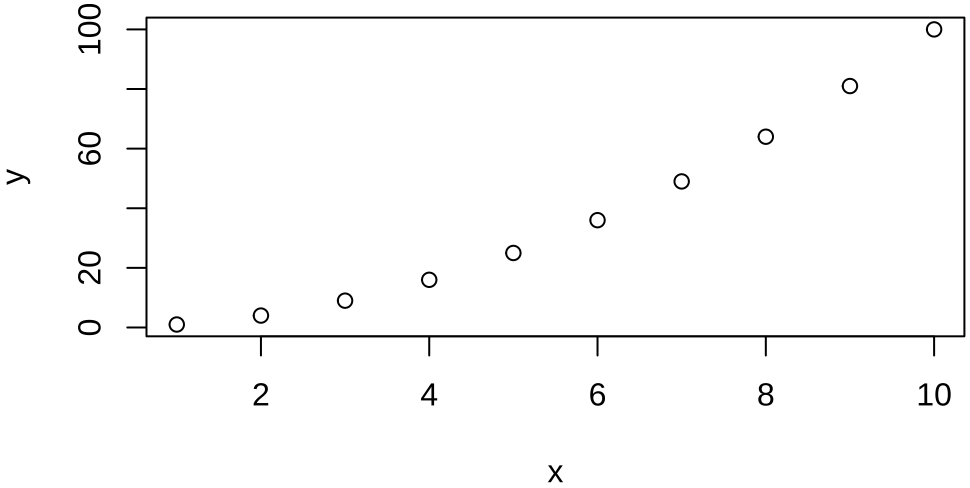
```
 [1]           1           4          27         256        3125       46656
 [7]      823543    16777216   387420489 10000000000
```

As we can see, since I didn't "store" this value as a variable in *R* using `<-`, the value is not in the environment.

3.3 Plotting

Now, let's try creating a plot. This is easy in *R*, as we just use the command `plot`.

```
plot(x = x, y = y)
```



By specifying the `y` and `x` components in `plot`, we can quickly generate a point plot. We can alter the visual parameters of this plot using a few different commands. I will outline these below with inline notes. Inline notes in the code can be made by using a `#` symbol before them, which basically tells *R* to ignore everything after the `#`. For example:

```
print("Test")
```

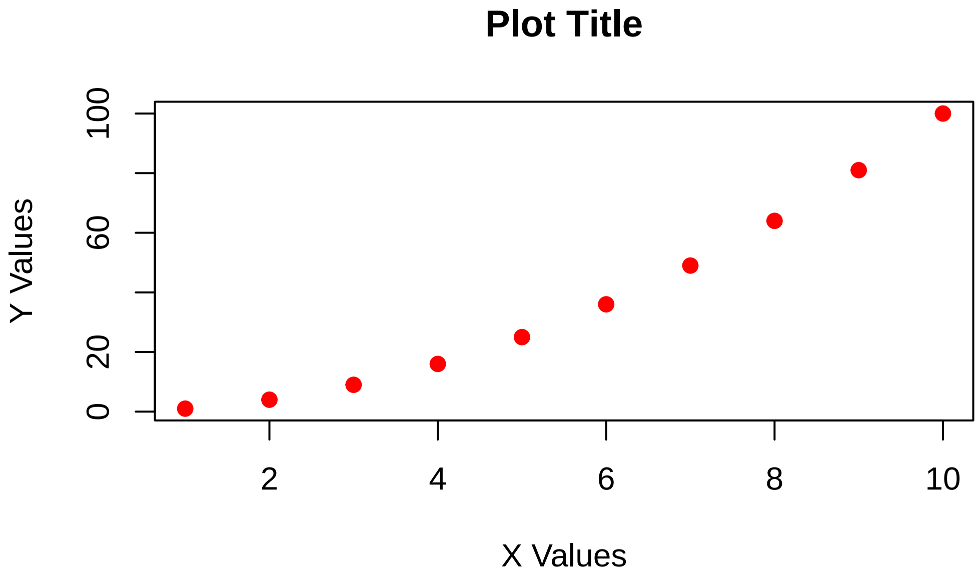
```
[1] "Test"
```

```
# print("Test 2")
```

This prints the word `Test`, but doesn't print `Test 2`.

Now let's make the plot with some new visual parameters.

```r
plot(x = x, # specify x values
     y = y, # specify y values
     ylab = "Y Values", # specify Y label
     xlab = "X Values", # specify X label
     main = "Plot Title", # specify main title
     pch = 19, # adjust point style
     col = "red") # make points red
```



3.4 Tab complete

RStudio allows for "tab-completing" while typing code. Tab-completing is a way of typing the first part of a command, variable name, or file name and hitting "tab" to show all options with that spelling. You should use tab completing because it:

- reduces spelling mistakes
- reduces filepath mistakes
- increases the speed at which you code
- provides help with specific functions

3.5 Help

At any point in *R*, you can look up "help" for a specific function by typing `?functionname`. Try this on your computer with the following:

```
?mean
```

4 Working with data

Throughout this course, we are going to have to work with datasets that are from our book or other sources. Here, we are going to work through an example dataset. First, we need to install *libraries*. A *library* is a collated, pre-existing batch of code that is designed to assist with data analysis or to perform specific functions. These *libraries* make life a lot easier, and create short commands for completing relatively complex tasks.

In this class, there are two libraries that you will need *almost every week*! First, we need to install the libraries. The main libraries we need for this class are:

1. `cURL`: this package allows us to download things from URLs. We will be using this to download data files. *Otherwise, you will have to enter all data by hand!*
2. `tidyverse`: this package is actually a group of packages designed to help with data analysis, management, and visualization.

```
# run this code the first time ONLY
# does not need to be run every time you use R!

# curl allows for internet downloads
install.packages("curl")

# tidyverse has a bunch of packages in it!
# great for data manipulation
install.packages("tidyverse")

# if you ever need to update:
# leaving brackets open means "update everything"
update.packages()
```

After packages are installed, we will need to load them into our *R* environment. While we only need to `install.packages` once on our machine, we need to load libraries *every time we restart the program*!

```
library(curl)
```

```
Using libcurl 8.7.1 with LibreSSL/3.3.6
```

```r
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter()     masks stats::filter()
x dplyr::lag()        masks stats::lag()
x readr::parse_date() masks curl::parse_date()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

You should an output like the above. What this means is:

1. The core packages that comprise the tidyverse loaded successfully, and version numbers for each are shown.
2. The conflicts basically means that certain commands will not work as they used to because *R* has "re-learned" a particular word.

To clarify the conflicts, pretend that you can only know one definition of a word at a time. You may know the word "cola" as a type of soda pop or as a drink in general. However, in Spanish, "cola" refers to a line or a tail. While we can learn both of these definitions and know which one is which because of context, a computer can't do that. In *R*, we would then have to specify which "cola" we are refferring to. We do this by listing the package before the command; in this case, `english::cola` would mean a soda pop and `spanish::cola` would refer to a line or tail. If we just type `cola`, the computer will assume one of these definitions but not even consider the other.

We won't have to deal with conflicts much in this class, and I'll warn you (or help you) if there is a conflict.

4.1 Downloading data

Now, we need to download our first data set. These datasets are stored on GitHub. We are going to be looking at data from Dr. Cooper's dissertation concerning Afrotropical bird distributions (Cooper 2021). This website is in the data folder on this websites' GitHub page, accessible here.

```
# first, declare filepath
# I will try to give you the filepath for each assignment
# if not, check the URL pattern for the file

# create
ranges.url <- curl("https://raw.githubusercontent.com/jacobccooper/biol105_unk/main/datasets/
# read comma separated file (csv) into R memory
ranges <- read_csv(ranges.url)
```

```
Rows: 12 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): species
dbl (9): combined_current_km2, consensus_km2, bioclim_current_km2, 2050_comb...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Alternatively, we can use the operator %>% to simplify this process. %>% means "take whatever you got from the previous step and *pipe* it into the next step". So, the following does the exact same thing:

```
ranges <- curl("https://raw.githubusercontent.com/jacobccooper/biol105_unk/main/datasets/lacu
  read_csv()
```

```
Rows: 12 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): species
dbl (9): combined_current_km2, consensus_km2, bioclim_current_km2, 2050_comb...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Using the %>% is preferred as you can better set up a workflow and because it more closely mimics other coding languages, such as `bash`.

Let's view the data to see if it worked. We can use the command `head` to view the first few rows:

```r
head(ranges)
```

```
# A tibble: 6 x 10
  species                combined_current_km2 consensus_km2 bioclim_current_km2
  <chr>                                 <dbl>         <dbl>               <dbl>
1 Batis_diops                          25209.         6694.              19241.
2 Chamaetylas_poliophrys               68171.         1106.              68158.
3 Cinnyris_regius                      60939.        13305.              53627.
4 Cossypha_archeri                     27021.         6409.              11798.
5 Cyanomitra_alinae                    78680.        34320.              63381.
6 Graueria_vittata                      8770.          861.               8301.
# i 6 more variables: `2050_combined_km2` <dbl>, `2050_consensus_km2` <dbl>,
#   `2070_combined_km2` <dbl>, `2070_consensus_km2` <dbl>,
#   alltime_consensus_km2 <dbl>, past_stable_km2 <dbl>
```

We can perform a lot of summary statistics in *R*. Some of these we can view for multiple columns at once using `summary`.

```r
summary(ranges)
```

```
   species          combined_current_km2 consensus_km2     bioclim_current_km2
 Length:12          Min.   :  8770       Min.   :  861.3   Min.   :  3749  
 Class  :character  1st Qu.: 24800       1st Qu.: 4186.2   1st Qu.: 10924  
 Mode   :character  Median : 43654       Median : 7778.1   Median : 31455  
                    Mean   : 68052       Mean   :18161.8   Mean   : 42457  
                    3rd Qu.: 70798       3rd Qu.:18558.7   3rd Qu.: 62835  
                    Max.   :232377       Max.   :79306.6   Max.   :148753  
 2050_combined_km2 2050_consensus_km2 2070_combined_km2  2070_consensus_km2
 Min.   :  1832    Min.   :    0.0    Min.   :   550.3   Min.   :    0.0  
 1st Qu.:  6562    1st Qu.:  589.5    1st Qu.:  6583.8   1st Qu.:  311.4  
 Median : 26057    Median : 6821.9    Median : 24281.7   Median : 2714.6  
 Mean   : 33247    Mean   :14418.4    Mean   : 31811.0   Mean   : 8250.5  
 3rd Qu.: 40460    3rd Qu.:18577.1    3rd Qu.: 38468.9   3rd Qu.:10034.4  
 Max.   :132487    Max.   :79236.2    Max.   :129591.0   Max.   :53291.8  
 alltime_consensus_km2 past_stable_km2
 Min.   :    0.0       Min.   :   0.0  
 1st Qu.:  790.9       1st Qu.:   0.0  
 Median : 8216.8       Median :   0.0  
 Mean   :15723.3       Mean   : 127.3  
 3rd Qu.:19675.0       3rd Qu.:   0.0  
 Max.   :82310.5       Max.   :1434.8  
```

As seen above, we now have information for the following statistics for each variable:

- `Min` = minimum
- `1st Qu.` = 1st quartile
- `Median` = middle of the dataset
- `Mean` = average of the dataset
- `3rd Qu.` = 3rd quartile
- `Max.` = maximum

We can also calculate some of these statistics manually to see if we are doing everything correctly. It is easiest to do this by using predefined functions in *R* (code others have written to perform a particular task) or to create our own functions in *R*. We will do both to determine the average of `combined_current_km2`.

4.2 Subsetting data

First, we need to select only the column of interest. In *R*, we have two ways of subsetting data to get a particular column.

- `var[rows,cols]` is a way to look at a particular object (`var` in this case) and choose a specific combination of `row` number and `column` number (`col`). This is great if you know a specific index, but it is better to use a specific name.
- `var[rows,"cols"]` is a way to do the above but by using a specific column name, like `combined_current_km2`.
- `var$colname` is a way to call the specific column name directly from the dataset.

```
# using R functions

ranges$combined_current_km2
```

```
 [1]  25209.4  68171.2  60939.2  27021.3  78679.9   8769.9 232377.2  17401.4
 [9]  51853.5  35455.1  23570.3 187179.1
```

As shown above, calling the specific column name with `$` allows us to see only the data of interest. We can also save these data as an object.

```
current_combined <- ranges$combined_current_km2

current_combined
```

```
 [1]  25209.4  68171.2  60939.2  27021.3  78679.9   8769.9 232377.2  17401.4
 [9]  51853.5  35455.1  23570.3 187179.1
```

Now that we have it as an object, specifically a numeric vector, we can perform whatever math operations we need to on the dataset.

```
mean(current_combined)
```

```
[1] 68052.29
```

Here, we can see the mean for the entire dataset. However, we should always round values to the same number of decimal points as the original data. We can do this with `round`.

```
round(mean(current_combined),1) # round mean to one decimal
```

```
[1] 68052.3
```

Note that the above has a nested set of commands. We can write this exact same thing as follows:

```
# pipe mean through round
mean(current_combined) %>%
  round(1)
```

```
[1] 68052.3
```

Use the method that is easiest for you to follow!

We can also calculate the mean manually. The mean is $\frac{\sum_{i=1}^{n} x}{n}$, or the sum of all the values within a vector divided by the number of values in that vector.

```
# create function
# use curly brackets to denote function
# our data goes in place of "x" when finally run
our_mean <- function(x){
  sum_x <- sum(x) # sum all values in vector
  n <- length(x) # get length of vector
  xbar <- sum_x/n # calcualte mean
  return(xbar) # return the value outside the function
}
```

Let's try it.

```
our_mean(ranges$combined_current_km2)
```

```
[1] 68052.29
```

As we can see, it works just the same as `mean`! We can round this as well.

```
our_mean(ranges$combined_current_km2) %>%
  round(1)
```

```
[1] 68052.3
```

5 Your turn!

With a partner or on your own, try to do the following:

1. Create an *RMarkdown document* that will save as an `.html`.
2. Load the data, as shown here, and print the summary statistics in the document.
3. Calculate the value of `combined_current_km2` divided by `2050_combined_km2` and print the results.

Let me know if you have any issues.

6 Descriptive Statistics

6.1 Purposes of descriptive statistics

Descriptive statistics enable researchers to quickly and easily examine the “behavior” of their datasets, identifying potential errors and allowing them to observe particular trends that may be worth further analysis. Here, we will cover how to calculate descriptive statistics for multiple different datasets, culminating in an assignment covering these topics.

6.2 Preparing *R*

As with every week, we will need to load our relevant packages first. This week, we are using the following:

```r
# allows for internet downloading
library(curl)
```

```
Using libcurl 8.7.1 with LibreSSL/3.3.6
```

```r
# enables data management tools
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.3     v tidyr     1.3.1
v purrr     1.0.2

-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter()     masks stats::filter()
x dplyr::lag()        masks stats::lag()
x readr::parse_date() masks curl::parse_date()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
```

6.3 Downloading the data

For the example this week, we will be using the `starbucks` dataset, describing the number of drinks purchased during particular time periods during the day.

```
starbucks <- curl("https://raw.githubusercontent.com/jacobccooper/biol105_unk/main/datasets/
  read_csv()
```

```
Rows: 9 Columns: 2
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): Hour
dbl (1): Frap_Num

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

6.4 Descriptive statistics

Descriptive statistics are statistics that help us understand the shape and nature of the data on hand. These include really common metrics such as *mean*, *median*, and *mode*, as well as more nuanced metrics like *quartiles* that help us understand if there is any *skew* in the dataset. (*Skew* refers to a bias in the data, where more data points lie on one side of the distribution and there is a long *tail* of data in the other direction).

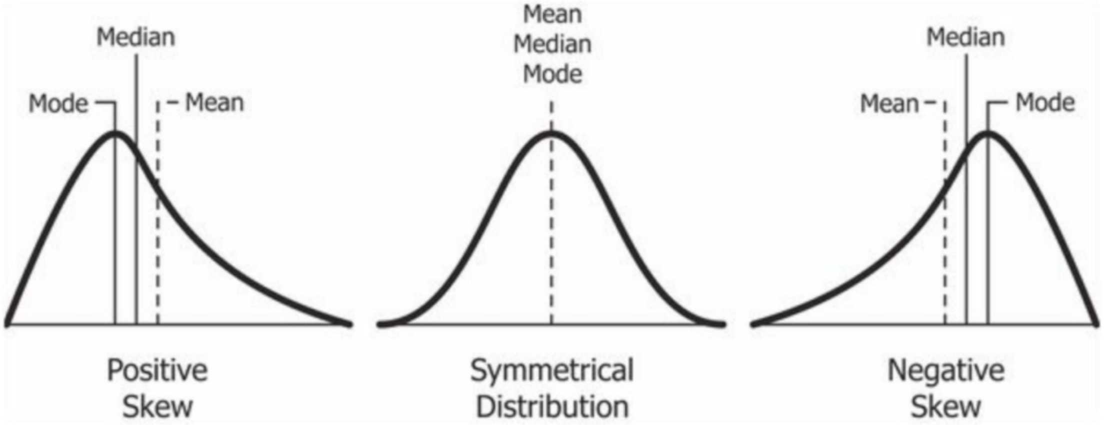


Figure 6.1: Examples of skew compared to a symmetrical, non-skewed distribution. Source: machinelearningparatodos.com

Note above that the relative position of the *mean*, *median*, and *mode* can be indicative of skew. Please also note that these values will rarely be exactly equal "in the real world", and thus you need to weigh differences against the entire dataset when assessing skew. There is a lot of nuance like this in statistics; it is not always an "exact" science, but sometimes involves judgment calls and assessments based on what you observe in the data.

Using the `starbucks` dataset, we can look at some of these descriptive statistics to understand what is going on.

6.4.1 Mean

The mean is the "average" value within a set of data, specifically, the sum of all values divided by the length of those values: $\frac{\sum_{i=1}^{n} x}{n}$.

```
head(starbucks)
```

```
# A tibble: 6 x 2
  Hour      Frap_Num
  <chr>        <dbl>
1 0600-0659        2
2 0700-0759        3
3 0800-0859        2
4 0900-0959        4
5 1000-1059        8
6 1100-1159        7
```

Here, we are specifically interested in the number of frappuccinos.

```
# get vector of frappuccino number
fraps <- starbucks$Frap_Num

# get mean of vector
mean(fraps)
```

```
[1] 6.222222
```

Note that the above should be rounded to a whole number, since we were given the data in whole numbers!

```
mean(fraps) %>%
  round(0)
```

```
[1] 6
```

We already covered calculating the average manually in our previous tutorial, but we can do that here as well:

```
# sum values
# divide by n, length of vector
# round to 0 places
round(sum(fraps)/length(fraps),0)
```

```
[1] 6
```

6.4.2 Median

The median is also known as the 50th percentile, and is the midpoint of the data when ordered from least to greatest. If there are an even number of data points, then it is the average point between the two center points. For odd data, this is the $\frac{n+1}{2}$th observation. For even data, since we need to take an average, this is the $\frac{\frac{n}{2}+(\frac{n}{2}+1)}{2}$. You should be able to do these by hand and by using a program.

```
median(fraps)
```

```
[1] 4
```

Now, to calculate by hand:

```
length(fraps)
```

```
[1] 9
```

We have an odd length.

```
# order gets the order
order(fraps)
```

```
[1] 1 3 7 2 4 6 5 9 8
```

```r
# [] tells R which elements to put where
frap_order <- fraps[order(fraps)]

frap_order
```

```
[1]  2  2  2  3  4  7  8 13 15
```

```r
# always use parentheses
# make sure the math maths right!
(length(frap_order)+1)/2
```

```
[1] 5
```

Which is the fifth element in the vector?

```r
frap_order[5]
```

```
[1] 4
```

Now let's try it for an even numbers.

```r
# remove first element
even_fraps <- fraps[-1]

even_fraps_order <- even_fraps[order(even_fraps)]

even_fraps_order
```

```
[1]  2  2  3  4  7  8 13 15
```

```r
median(even_fraps)
```

```
[1] 5.5
```

Now, by hand: $\frac{\frac{n}{2}+(\frac{n}{2}+1)}{2}$.

```r
n <- length(even_fraps_order)

# get n/2 position from vector
m1 <- even_fraps_order[n/2]
# get n/2+1 position
m2 <- even_fraps_order[(n/2)+1]

# add these values, divide by two for "midpoint"
med <- (m1+m2)/2

med
```

```
[1] 5.5
```

As we can see, these values are equal!

6.4.3 Mode

There is no default method for finding the mode in *R*. However, websites like Statology provide wraparound functions.

```r
# Statology function
# define function to calculate mode
find_mode <- function(x) {
  # get unique values from vector
  u <- unique(x)
  # count number of occurrences for each value
  tab <- tabulate(match(x, u))
  # return the value with the highest count
  u[tab == max(tab)]
}

find_mode(fraps)
```

```
[1] 2
```

We can also do this by hand, by counting the number of occurrences of each value. This can be done in a stepwise fashion using commands in the above function.

```r
# unique counts
u <- unique(fraps)
u
```

```
[1]  2  3  4  8  7 15 13
```

```r
# which elements match
match(fraps,u)
```

```
[1] 1 2 1 3 4 5 1 6 7
```

```r
# count them
tab <- match(fraps,u) %>%
  tabulate()

tab
```

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

Get the highest value.

```r
u[tab==max(tab)]
```

```
[1] 2
```

Notice this uses `==`. This is a logical argument that means "is equal to" or "is the same as". For example:

```r
2 == 2
```

```
[1] TRUE
```

These values are the same, so `TRUE` is returned.

```r
2 == 3
```

```
[1] FALSE
```

These values are unequal, so `FALSE` is returned. *R* will read `TRUE` as `1` and `FALSE` as `ZERO`, such that:

```
sum(2==2)
```

```
[1] 1
```

and

```
sum(2==3)
```

```
[1] 0
```

This allows you to find how many arguments match your condition quickly, and even allows you to subset based on these indices as well. Keep in mind you can use greater than `<`, less than `>`, greater than or equal to `<=`, less than or equal to `>=`, is equal to `==`, and is not equal to `!=` to identify numerical relationships. Other logical arguments include:

- `&`: both conditions must be `TRUE` to match (e.g., `c(10,20) & c(20,10)`). Try the following as well: `fraps < 10 & fraps > 3`.
- `&&`: and, but works with single elements and allows for better parsing. Often used with `if`. E.g., `fraps < 10 && fraps > 3`. This will not work on our multi-element `frap` vector.
- `|`: or, saying at least one condition must be true. Try: `fraps > 10 | fraps < 3`.
- `||`: or, but for a single element, like `&&` above.
- `!`: not, so "not equal to" would be `!=`.

6.4.4 Other quartiles

We also use the 25th percentile and the 75th percentile to understand data distributions. These are calculated similar to the above, but the bottom quartile is only $\frac{1}{4}$ of the way between values and the 75th quartile is $\frac{3}{4}$ of the way between values. We can use the *R* function `quantile` to calculate these.

```
quantile(frap_order)
```

```
  0%  25%  50%  75% 100% 
   2    2    4    8   15 
```

We can specify a quantile as well:

```r
quantile(frap_order, 0.75)
```

```
75%
  8
```

We can also calculate these metrics by hand.

References

Cooper, J. C. (2021). Biogeographic and Ecologic Drivers of Avian Diversity. [Online.] Available at https://doi.org/10.6082/uchicago.3379.