204A Lab Week 2

2023-10-05

Our first task in this lab is to learn about file paths and how to load a file. We will be working with three file types: comma separated values (.csv) files, excel file (.xlsx) and text (.txt) files. In the Lab 2 Materials, you will find the data files of both types and you should download these before we get started. It might also be helpful to download the files to a separate folder and find the

Today's lab code (and most of the lab code in this class) was taken and modified from previous versions of past TAs of this course Josue Rodriguez and Alexander Baxter. We also use open source data from the following publication:

Dettmer AM, Rosenberg KL, Suomi SJ, Meyer JS, Novak MA (2015). Associations between Parity, Hair Hormone Profiles during Pregnancy and Lactation, and Infant Development in Rhesus Monkeys (Macaca mulatta). PLOS ONE 10(7): e0131692. https://doi.org/10.1371/journal.pone.0131692

Some great references R:

- General R stuff: R for Data Science by Hadley Wickham and Garret Grolemund
- The dplyr package: documentation
- The ggplot2 package: documentation

Reading in Data in R

Reading a CSV data file

Today we are going to learn how to read comma separated values (CSV) files into R.

CSV files are like a spreadsheet, except rows are represented by a new line, and columns are represented by commas.

The first line of a CSV file usually contains the column names.

For the lab data we will use this week, the first three lines of the csv file look like this:

```
Subject,Sex,Mother_Parity,Hair_Cortisol,Age,Irritability,Consolability
N001,Male,Multiparous,70.50,202,1.83,1.33
N006,Male,Multiparous,71.20,221,1.63,1.67
```

To read in .csv files, we will use the `read.csv()` function. This function has several arguments, but the primary one to use is 'file'. This is the file path where the data are saved on your computer. So before we use the `read.csv()` function, we will need to determine the file path. There are several ways to do this.

Option 1: Look up the file path manually

To do this, you can use the file.choose function.

```
file.choose()
# In the window that opens, navigate to the file path, and click "open". This will return the file path
```

```
# Then, copy and paste the output path above into read.csv()

# NOTE: I showed you this process in several steps for the sake of illustration.
```

```r
#  In reality, you can do this process in a single line of code, if you know the file path.
# The following line of code would give you the same result (at least for me on my laptop, where I have

lab_data <- read.csv("/Users/rbat/Library/CloudStorage/Box-Box/204A/Lab 2 (Rohit)/data_lab_week2.csv")
```

```r
lab_data
```

```
##    Subject    Sex Mother_Parity Hair_Cortisol Age Irritability Consolability
## 1     N001   Male   Multiparous          70.5 202         1.83          1.33
## 2     N006   Male   Multiparous          71.2 221         1.63          1.67
## 3     N011 Female   Multiparous          90.0 122         1.17          1.00
## 4     N012   Male   Multiparous          87.6 165         0.88          1.75
## 5     N064 Female   Multiparous          88.0 122         1.63          1.67
## 6     N062   Male   Multiparous          74.3 178         1.13          1.25
## 7     N049 Female   Multiparous          65.5  80         1.50          1.25
## 8     N021 Female   Multiparous          82.0 174         1.00          1.33
## 9     N022   Male   Multiparous          64.4 190         1.33          1.50
## 10    N024 Female   Multiparous          72.9 185         1.50          1.38
## 11    N030   Male   Multiparous          90.4  85         1.75          1.75
## 12    N075   Male   Multiparous          67.3 214         1.33          1.17
## 13    N035   Male   Multiparous          62.5 163         1.83          1.83
## 14    N037 Female   Multiparous          73.1 213         0.88          0.75
## 15    N067   Male   Primiparous          67.9 196         1.50          1.38
## 16    N132   Male   Primiparous         102.7 193         1.25          1.50
## 17    N133 Female   Primiparous         110.2 114         1.83          1.83
## 18    N070 Female   Primiparous          69.9 152         1.50          1.50
## 19    N068 Female   Primiparous         117.3 201         1.33          1.00
## 20    N071 Female   Primiparous          88.1 165         1.75          1.63
## 21    N069   Male   Primiparous          74.7 208         1.50          1.50
## 22    N130 Female   Primiparous          72.8 104         1.75          1.75
## 23    N057 Female   Primiparous          59.8  93         1.63          1.88
## 24    N065   Male   Primiparous          78.4 180         1.75          1.75
## 25    N058   Male   Primiparous          71.3  92         1.67          1.67
## 26    N059   Male   Primiparous          73.4 127         1.75          2.00
```

You could also do this by saving the file path as an object. An object is some type of data that is saved with a name in your R environment. You can save any thing as an object.

For example:

```r
an_object <- 2

# Now, when you call `an_object`, it returns the number 2.
an_object
```

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

Let's try to read the data in again, this time saving the file path as an object. We can use the `file.choose()` function to automate this process:

```r
path <-  file.path("/", "Users", "rbat", "Library", "CloudStorage", "Box-Box", "204A", "Lab 2 (Rohit)",
path # This is the saved file path to where the data are stored on your computer. The path for me will
```

```
## [1] "//Users/rbat/Library/CloudStorage/Box-Box/204A/Lab 2 (Rohit)/data_lab_week2.csv"
```

```r
# usually this function will add the forward slash between the directories
# Now, instead of the file path, use the read.csv() function to read the data into R.
lab_data <- read.csv(file = path)
```

There are also shortcuts in windows and mac operating systems to get the file path:

1. For windows, you right click and press the shift key (https://www.timeatlas.com/copy-file-path/)
2. For mac, you right click and press the option key (https://setapp.com/how-to/how-to-find-the-path-of-a-file-in-mac)

Option 2: Set a Working Directory

We went over this briefly last week, but here is a review.

The working directory is the "default folder" on your computer that R will use to read and write data.

All data will be read and written from this working folder, unless you specify otherwise. If you do not set a working directory, then R will assume that this is the your "home" directory.

If the file is an .Rmd instead of .R, then the working directory will be the same as the .Rmd file you opened.

First, lets check where your working directory is set:

```
getwd()
```

```
## [1] "/Users/rbat/Library/CloudStorage/Box-Box/204A/Lab 2 (Rohit)"
```

Because this the working directory is the same folder where I have my .csv file saved, I could also read in the data using this line of code:

```
lab_data <- read.csv("data_lab_week2.csv")
```

This is equivalent to the code I had for option 1.

Notice how you could piece together the same file path that is in option 1 by putting together the working directory and the file specification in this code.

I can only read in files from the working directory that actually exist in the working directory. For example, this does not work:

```
lab_data <- read.csv("hw_data_week2.csv")
```

```
## Warning in file(file, "rt"): cannot open file 'hw_data_week2.csv': No such file
## or directory

## Error in file(file, "rt"): cannot open the connection
```

If you ever want to change the working directory, you can set it using the `setwd()` function. For example, if I wanted to work from my desktop, I could set that as the working directory like this:

```
setwd("~/Desktop")

# Notice that this does not work any more:
lab_data <- read.csv("data_lab_week2.csv")
```

```
## Warning in file(file, "rt"): cannot open file 'data_lab_week2.csv': No such
## file or directory

## Error in file(file, "rt"): cannot open the connection
```

I personally prefer to use Option 1 over Option 2, so I can always find my files if I ever need to return to my code after a long time. But you are welcome to do it any way you would like.

I am going to reset the working directory back to how it was before:

```r
setwd("/Users/rbat/Library/CloudStorage/Box-Box/204A/Lab 2 (Rohit)")
```

Reading an Excel data file

What if you only have an Excel file, and you do not have a CSV file?

You have a few options in this case.

Option 1: Simply open your data in Excel, and save the data as a CSV file.

Note: When you do this, all formulas will be converted to values, and all formatting will be lost. This is OK as long as you have the original Excel file saved. There are packages that will read data directly from Excel, but I prefer the CSV method.

Option 2: Use read_excel(), from the readxl package.

Note: A package is a series of functions and formulas that are not part of the basic R software. You can often find many useful functions in packages.

```r
if (!require("readxl")) install.packages("readxl") # This will install the package if it is not already
```

```
## Loading required package: readxl
```

```r
# This will load the package so that you can use the functions from it.
library(readxl)
```

```r
# Notice I did not specify the full file path, because this excel file is already located in my working
# If I was opening a file located in a different location (other than the working directory), I would n
# the entire file path.
lab_data_xl <- read_excel(path = "excel_data_lab_week2.xlsx")
```

If you have more than one sheet in your excel file, you can specify which tab to read in with the sheet argument. You do not need to specify this if there is only one sheet (as in the example above), but you can if you would like to.

```r
lab_data_xl <- read_excel(path = "excel_data_lab_week2.xlsx", sheet = 1)

lab_data_xl <- read_excel(path = "excel_data_lab_week2.xlsx", sheet = "Monkey Data")
```

Read a text data file

Sometimes our data might also be in text data format. This is another type of file type and base R has a function to read this file type called `read.table()`

```r
lab_data_txt <- read.table("/Users/rbat/Library/CloudStorage/Box-Box/204A/Lab 2 (Rohit)/data_lab_week2.

#Let's view the data

View(lab_data_txt)
```

Something strange happened: The first row is not the row of data but rather column names. This is because in the `read.table()` function the default argument (more on it below) of header is set to be FALSE. So we should change that:

```r
lab_data_txt <- read.table("/Users/rbat/Library/CloudStorage/Box-Box/204A/Lab 2 (Rohit)/data_lab_week2.

#Let's view the data
```

```
View(lab_data_txt) # Now the file looks correctly loaded!
```

It is important we use the correct function for the file type. Try using a `read.csv` function for a text data file and see what happens?

Default Arguments

The vast majority of functions in R have at least one “argument”. An argument is an input to the function that determines the behavior of the function.

Some arguments need to be explicitly specified, or else the function will not work and it will return an error. But in many cases, functions have default inputs that will be used if you don’t specify an argument. These are called *default arguments*. In other words, a default argument is what the function will do by default unless you tell it to specifically do something else.

Lets return to the `read.csv()`function for a demonstration.

In the code above, we ran:

```
lab_data <- read.csv("data_lab_week2.csv")
```

But this is implicitly running

```
lab_data <- read.csv(file = "data_lab_week2.csv")
```

In this case, the only argument that we specified was `file =`. This is because the file argument in `read.csv()` does not have a default argument (i.e., we need to tell the function what file we want to read in, it has no default for ‘file’ if we don’t specify it).

Although there are many other arguments for `read.csv()`, we did not need to give inputs to these other arguments because all the other ones had default settings.

Lets take a closer look.

Looking at Documentation

First, lets look at the documentation for `read.csv()`, to see what arguments there are.

There are several ways to look at documentation. The first way is to use the `?` operator.

```
# To pull up the documentation for read.csv(), run the line of code below:
?read.csv
```

Notice that each argument except ‘file’, there is an `=` . This indicates there is a default argument. What follows the `=` is the default argument for that specific default.

If there is no `=` , that means there is no default for that argument, and if you do not give that argument and input the function will not run (i.e., ‘file’ in this example)

Default Arguments

I will demonstrate how default arguments work by showing you what happens when we vary some of the arguments in `read.csv()`.

We will play with the following arguments for `read.csv()`:

- header: This is a TRUE or FALSE argument that tells `read.csv()`whether the first row of values are the headers for the data. The default is TRUE.

- sep: This tells `read.csv()` what symbol is used to separate unique values in the CSV file. The default is a comma (hence, “comma separated values”).

Because of default arguments, the following lines of code are equivalent:

```
path <- "data_lab_week2.csv"

lab_data <- read.csv(file = path)
head(lab_data)
```

```
##   Subject    Sex Mother_Parity Hair_Cortisol Age Irritability Consolability
## 1    N001   Male   Multiparous          70.5 202         1.83          1.33
## 2    N006   Male   Multiparous          71.2 221         1.63          1.67
## 3    N011 Female   Multiparous          90.0 122         1.17          1.00
## 4    N012   Male   Multiparous          87.6 165         0.88          1.75
## 5    N064 Female   Multiparous          88.0 122         1.63          1.67
## 6    N062   Male   Multiparous          74.3 178         1.13          1.25
```

```
lab_data <- read.csv(file = path, header = TRUE, sep = ",")
head(lab_data)
```

```
##   Subject    Sex Mother_Parity Hair_Cortisol Age Irritability Consolability
## 1    N001   Male   Multiparous          70.5 202         1.83          1.33
## 2    N006   Male   Multiparous          71.2 221         1.63          1.67
## 3    N011 Female   Multiparous          90.0 122         1.17          1.00
## 4    N012   Male   Multiparous          87.6 165         0.88          1.75
## 5    N064 Female   Multiparous          88.0 122         1.63          1.67
## 6    N062   Male   Multiparous          74.3 178         1.13          1.25
```

These lines are the same because the default argument for ‘header’ is TRUE, and the default for `sep` is “,”, so they don’t need to be specified unless you want to change that argument.

Let’s see what happens when we start to change the default arguments:

```
## If we set header = FALSE, then the first row of values will NOT be used as the variable names.
lab_data_no_header <- read.csv(file = path, header = FALSE, sep = ",")

head(lab_data_no_header)
```

```
##        V1     V2            V3            V4  V5           V6            V7
## 1 Subject    Sex Mother_Parity Hair_Cortisol Age Irritability Consolability
## 2    N001   Male   Multiparous         70.50 202         1.83          1.33
## 3    N006   Male   Multiparous         71.20 221         1.63          1.67
## 4    N011 Female   Multiparous         90.00 122         1.17          1.00
## 5    N012   Male   Multiparous         87.60 165         0.88          1.75
## 6    N064 Female   Multiparous         88.00 122         1.63          1.67
```

Argument

If you are listing arguments without specifying the argument, R will assume that you are specifying arguments in the order the arguments are listed in the documentation. So, the following lines of code are the same:

```
lab_data <- read.csv(file = path, header = TRUE, sep = ",")
lab_data <- read.csv(path, TRUE, ",")
```

If you don’t indicate the name of the arguments, it gives an error because it thinks the comma is some kind of path to a file on the computer, and obviously there is no path to a file with only a single comma.

```r
lab_data <- read.csv(",", TRUE, path)
```

```
## Warning in file(file, "rt"): cannot open file ',': No such file or directory

## Error in file(file, "rt"): cannot open the connection
```

However, if you specify each individual argument, then the order does not matter. Hence, the following line of code is the same as the two in the chunk above.

```r
read.csv(file = path,   header = TRUE,  sep = ",")
read.csv(sep = ",",     file = path,    header = TRUE)
read.csv(header = TRUE, sep = ",",      file = path)
```

Calling and Subsetting Data in R

Let's take a look at today's lab data:

```r
colnames(lab_data)
```

```
## [1] "Subject"       "Sex"           "Mother_Parity" "Hair_Cortisol"
## [5] "Age"           "Irritability"  "Consolability"
```

The data come from a study examining neonatal development in rhesus monkeys. We will be working with the following variables:

- `Subject`: The ID for the infant monkey
- `Sex`: Monkey's biological sex
- `Mother_Parity`: Whether the monkey was born to a first-time mother (primiparous) or a mother who had reared an offspring before (multiparous)
- `Hair_Cortisol`: Concentrations of Cortisol assayed from hair samples
- `Age`: The infant monkeys' age at the time of sample collection
- `Irritability`: A measure of how irritable the infant is, based on temperament assessment
- `Consolability`: A measure of how consolable the infant is, based on temperament assessment

Now that we have our data imported into R, we want to do stuff with it. But how?

By simply typing and running the object name, you can call the data frame.

```r
lab_data
```

```
##    Subject    Sex Mother_Parity Hair_Cortisol Age Irritability Consolability
## 1     N001   Male   Multiparous          70.5 202         1.83          1.33
## 2     N006   Male   Multiparous          71.2 221         1.63          1.67
## 3     N011 Female   Multiparous          90.0 122         1.17          1.00
## 4     N012   Male   Multiparous          87.6 165         0.88          1.75
## 5     N064 Female   Multiparous          88.0 122         1.63          1.67
## 6     N062   Male   Multiparous          74.3 178         1.13          1.25
## 7     N049 Female   Multiparous          65.5  80         1.50          1.25
## 8     N021 Female   Multiparous          82.0 174         1.00          1.33
## 9     N022   Male   Multiparous          64.4 190         1.33          1.50
## 10    N024 Female   Multiparous          72.9 185         1.50          1.38
## 11    N030   Male   Multiparous          90.4  85         1.75          1.75
## 12    N075   Male   Multiparous          67.3 214         1.33          1.17
## 13    N035   Male   Multiparous          62.5 163         1.83          1.83
## 14    N037 Female   Multiparous          73.1 213         0.88          0.75
## 15    N067   Male   Primiparous          67.9 196         1.50          1.38
## 16    N132   Male   Primiparous         102.7 193         1.25          1.50
## 17    N133 Female   Primiparous         110.2 114         1.83          1.83
## 18    N070 Female   Primiparous           69.9 152          1.50           1.50
## 19    N068 Female   Primiparous          117.3 201          1.33           1.00
## 20    N071 Female   Primiparous           88.1 165          1.75           1.63
## 21    N069   Male   Primiparous           74.7 208          1.50           1.50
## 22    N130 Female   Primiparous           72.8 104          1.75           1.75
## 23    N057 Female   Primiparous           59.8  93          1.63           1.88
## 24    N065   Male   Primiparous           78.4 180          1.75           1.75
## 25    N058   Male   Primiparous           71.3  92          1.67           1.67
## 26    N059   Male   Primiparous           73.4 127          1.75           2.00
```

By using the $ operator, you can have one specific column of the data called. For example, lets look at the first variable in the data frame.

```
lab_data$Subject
```

```
##  [1] "N001" "N006" "N011" "N012" "N064" "N062" "N049" "N021" "N022" "N024"
## [11] "N030" "N075" "N035" "N037" "N067" "N132" "N133" "N070" "N068" "N071"
## [21] "N069" "N130" "N057" "N065" "N058" "N059"
```

This was one way of getting some portion of the data or what we call as subsetting the data.

Subsetting Data with base R

There are many ways of subsetting data. In this lab module, we will learn how to work with data in base R and in next module, we will use another way of subsetting data from the dplyr package.

You can also use brackets to call or subset data. This is done in the format of `dataframe[rows,columns]`. If either rows or columns are left blank, then all the rows or columns are returned. Rows and columns can be subsetted by row/column number or name:

```
head(lab_data)
```

```
##   Subject    Sex Mother_Parity Hair_Cortisol Age Irritability Consolability
## 1    N001   Male   Multiparous          70.5 202         1.83          1.33
## 2    N006   Male   Multiparous          71.2 221         1.63          1.67
## 3    N011 Female   Multiparous          90.0 122         1.17          1.00
## 4    N012   Male   Multiparous          87.6 165         0.88          1.75
## 5    N064 Female   Multiparous          88.0 122         1.63          1.67
## 6    N062   Male   Multiparous          74.3 178         1.13          1.25
```

```
lab_data[, 1] # This returns all rows and the first column of data
```

```
##  [1] "N001" "N006" "N011" "N012" "N064" "N062" "N049" "N021" "N022" "N024"
## [11] "N030" "N075" "N035" "N037" "N067" "N132" "N133" "N070" "N068" "N071"
## [21] "N069" "N130" "N057" "N065" "N058" "N059"
```

```
lab_data[, "Subject"] # You can also use column names
```

```
##  [1] "N001" "N006" "N011" "N012" "N064" "N062" "N049" "N021" "N022" "N024"
## [11] "N030" "N075" "N035" "N037" "N067" "N132" "N133" "N070" "N068" "N071"
## [21] "N069" "N130" "N057" "N065" "N058" "N059"
```

If you ever want to call more than one variable at a time, then the $ operator is no longer useful. The following lines of code return the same thing:

```
lab_data[ , c(1,2,3)] # the c() function can be used to create a vector
```

```
##    Subject    Sex Mother_Parity
## 1     N001   Male   Multiparous
## 2     N006   Male   Multiparous
## 3     N011 Female   Multiparous
## 4     N012   Male   Multiparous
## 5     N064 Female   Multiparous
## 6     N062   Male   Multiparous
## 7     N049 Female   Multiparous
## 8     N021 Female   Multiparous
## 9     N022   Male   Multiparous
## 10    N024 Female   Multiparous
## 11    N030   Male   Multiparous
## 12    N075   Male   Multiparous
## 13    N035   Male   Multiparous
## 14    N037 Female   Multiparous
## 15    N067   Male   Primiparous
## 16    N132   Male   Primiparous
## 17    N133 Female   Primiparous
## 18    N070 Female   Primiparous
## 19    N068 Female   Primiparous
## 20    N071 Female   Primiparous
## 21    N069   Male   Primiparous
## 22    N130 Female   Primiparous
## 23    N057 Female   Primiparous
## 24    N065   Male   Primiparous
## 25    N058   Male   Primiparous
## 26    N059   Male   Primiparous
```

```
lab_data[ , 1:3] # a colon can be used to indicate "through" (i.e., 1 through 3, or 1 2 3)
```

```
##    Subject    Sex Mother_Parity
## 1     N001   Male   Multiparous
## 2     N006   Male   Multiparous
## 3     N011 Female   Multiparous
## 4     N012   Male   Multiparous
## 5     N064 Female   Multiparous
## 6     N062   Male   Multiparous
## 7     N049 Female   Multiparous
## 8     N021 Female   Multiparous
## 9     N022   Male   Multiparous
## 10    N024 Female   Multiparous
## 11    N030   Male   Multiparous
## 12    N075   Male   Multiparous
## 13    N035   Male   Multiparous
## 14    N037 Female   Multiparous
## 15    N067   Male   Primiparous
## 16    N132   Male   Primiparous
## 17    N133 Female   Primiparous
## 18    N070 Female   Primiparous
## 19    N068 Female   Primiparous
## 20    N071 Female   Primiparous
## 21    N069   Male   Primiparous
## 22    N130 Female   Primiparous
## 23    N057 Female   Primiparous
## 24    N065   Male   Primiparous
## 25    N058   Male   Primiparous
## 26    N059   Male   Primiparous
```

```
lab_data[ , c("Subject","Sex","Mother_Parity")] # In this example, you use the c() function to create a
```

```
##    Subject    Sex Mother_Parity
## 1     N001   Male   Multiparous
## 2     N006   Male   Multiparous
## 3     N011 Female   Multiparous
## 4     N012   Male   Multiparous
## 5     N064 Female   Multiparous
## 6     N062   Male   Multiparous
## 7     N049 Female   Multiparous
## 8     N021 Female   Multiparous
## 9     N022   Male   Multiparous
## 10    N024 Female   Multiparous
## 11    N030   Male   Multiparous
## 12    N075   Male   Multiparous
## 13    N035   Male   Multiparous
## 14    N037 Female   Multiparous
## 15    N067   Male   Primiparous
## 16    N132   Male   Primiparous
## 17    N133 Female   Primiparous
## 18    N070 Female   Primiparous
## 19    N068 Female   Primiparous
## 20    N071 Female   Primiparous
## 21    N069   Male   Primiparous
## 22    N130 Female   Primiparous
## 23    N057 Female   Primiparous
## 24    N065   Male   Primiparous
## 25    N058   Male   Primiparous
## 26    N059   Male   Primiparous
```

Data Classes

Determining data classes in R

So far, you only knew that the variables we examined in the previous chunk were 'factor' or 'numeric' because I told you. The chunk below explains how to determine what class your data is.

```
head(lab_data)
```

```
##   Subject    Sex Mother_Parity Hair_Cortisol Age Irritability Consolability
## 1    N001   Male   Multiparous          70.5 202         1.83          1.33
## 2    N006   Male   Multiparous          71.2 221         1.63          1.67
## 3    N011 Female   Multiparous          90.0 122         1.17          1.00
## 4    N012   Male   Multiparous          87.6 165         0.88          1.75
## 5    N064 Female   Multiparous          88.0 122         1.63          1.67
## 6    N062   Male   Multiparous          74.3 178         1.13          1.25
```

```
# How do you determine what class of data you have? To do this, you use the class() function.

class(lab_data$Subject)
```

```
## [1] "character"
```

```
class(lab_data$Sex)
```

```
## [1] "character"
```

```
class(lab_data$Hair_Cortisol)
```

```
## [1] "numeric"
```

Typing this all out for each variable in a data frame can be tedious. One solution to checking the class of each variable in the data frame is the `str()` (structure) function

```
str(lab_data)
```

```
## 'data.frame':    26 obs. of  7 variables:
##  $ Subject      : chr  "N001" "N006" "N011" "N012" ...
##  $ Sex          : chr  "Male" "Male" "Female" "Male" ...
##  $ Mother_Parity: chr  "Multiparous" "Multiparous" "Multiparous" "Multiparous" ...
##  $ Hair_Cortisol: num  70.5 71.2 90 87.6 88 74.3 65.5 82 64.4 72.9 ...
##  $ Age          : int  202 221 122 165 122 178 80 174 190 185 ...
##  $ Irritability : num  1.83 1.63 1.17 0.88 1.63 1.13 1.5 1 1.33 1.5 ...
##  $ Consolability: num  1.33 1.67 1 1.75 1.67 1.25 1.25 1.33 1.5 1.38 ...
```

Different types of data classes

There are many data classes in R. The ones we tend to use the most are:

- numeric (floats and integers)
- strings
- factors (like string but with predetermined categories)
- booleans or logicals (TRUE/FALSE)

Vector Level

Vectors are collection of values. They are the simplest of data structures in R, and are generally created with the `c()` function. Importantly, they can only contain a single class of data (e.g., all numerics, strings, etc.)

Examples of vectors are shown below:

```
c(2,8,10)
```

```
## [1]  2  8 10
```

```
c("Blue","Red","Yellow","Green")
```

```
## [1] "Blue"   "Red"    "Yellow" "Green"
```

What happens if we have multiple classes in a vector?

```
c(1, 3, "I love tacos")
```

```
## [1] "1"            "3"            "I love tacos"
```

One common data class is `character` data. `Character` data are simply text strings.

```
chr_example1 <- "This is just text"
class(chr_example1)
```

```
## [1] "character"
```

```
chr_example2 <- c("Josue","Statistics","Big Fan")
class(chr_example2)
```

```
## [1] "character"
```

We can make the (mixed) character vector above to numeric with:

```
tac <- c(1, 3, "I love tacos")
as.numeric(tac)
```

```
## Warning: NAs introduced by coercion

## [1]  1  3 NA
```

Now the last element will be NA because we cannot convert a character element to numeric.

Logical class data are values that are either `TRUE` or `FALSE` (T and F as shorthand).

```
logical_example <-c(TRUE,FALSE,FALSE,T,F)
class(logical_example)
```

```
## [1] "logical"
```

Data Level

Regardless of what class of data you have in your vector, vectors can be combined and organized into more complex data structures, in fact data frames in R are vectors stacked column-wise. Further, data structures are themselves a class of data.

```
vec1 <- c(1, 2, 3, 4)
vec2 <- c("a", "b", "c", "d")

class(vec1)
```

```
## [1] "numeric"
```

```
class(vec2)
```

```
## [1] "character"
```

```
df <- data.frame(vec1, vec2)
class(df)
```

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

Lists are also a combination of vectors. Each item of a list can vary in its class and length. You can create lists using the `list()` function. For example:

```
my_first_list <- list(vec1, vec2, 1, c("For me", "I really like corn"),df)
my_first_list
```

```
## [[1]]
## [1] 1 2 3 4
##
## [[2]]
## [1] "a" "b" "c" "d"
##
## [[3]]
## [1] 1
##
## [[4]]
## [1] "For me"             "I really like corn"
##
## [[5]]
##   vec1 vec2
## 1    1    a
## 2    2    b
## 3    3    c
## 4    4    d
```

```
class(my_first_list)
```

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

Coercing data between classes

In R lingo, coercing data means that you change it from one class to another. This is usually important when you first read in your data, because sometimes R does not read your data in as the correct class.

In the lab data, the variables are mostly read in correctly. However, for the sake of demonstration, lets coerce the `Subject` variable from a factor to character.

```
lab_data <- read.csv("data_lab_week2.csv")
```

By default, all strings are read in as class `character`

```
str(lab_data)
```

```
## 'data.frame':    26 obs. of  7 variables:
##  $ Subject      : chr  "N001" "N006" "N011" "N012" ...
##  $ Sex          : chr  "Male" "Male" "Female" "Male" ...
##  $ Mother_Parity: chr  "Multiparous" "Multiparous" "Multiparous" "Multiparous" ...
##  $ Hair_Cortisol: num  70.5 71.2 90 87.6 88 74.3 65.5 82 64.4 72.9 ...
##  $ Age          : int  202 221 122 165 122 178 80 174 190 185 ...
##  $ Irritability : num  1.83 1.63 1.17 0.88 1.63 1.13 1.5 1 1.33 1.5 ...
##  $ Consolability: num  1.33 1.67 1 1.75 1.67 1.25 1.25 1.33 1.5 1.38 ...
```

To coerce data to a factor, use `as.factor()`

```
as.factor(lab_data$Subject)
```

```
##  [1] N001 N006 N011 N012 N064 N062 N049 N021 N022 N024 N030 N075 N035 N037 N067
## [16] N132 N133 N070 N068 N071 N069 N130 N057 N065 N058 N059
## 26 Levels: N001 N006 N011 N012 N021 N022 N024 N030 N035 N037 N049 N057 ... N133
```

Notice that this did not save our change. If you call the variable again, you will notice that it is still a character.

```
lab_data$Sex
```

```
##  [1] "Male"   "Male"   "Female" "Male"   "Female" "Male"   "Female" "Female"
##  [9] "Male"   "Female" "Male"   "Male"   "Male"   "Female" "Male"   "Male"
## [17] "Female" "Female" "Female" "Female" "Male"   "Female" "Female" "Male"
## [25] "Male"   "Male"
```

To change the class, you need to overwrite the variable:

```
lab_data$Sex <- as.factor(lab_data$Sex)
```

Now when you call the variable it is a factor:

```
lab_data$Sex
```

```
##  [1] Male   Male   Female Male   Female Male   Female Female Male   Female
## [11] Male   Male   Male   Female Male   Male   Female Female Female Female
## [21] Male   Female Female Male   Male   Male
## Levels: Female Male
```

```r
class(lab_data$Sex)
```

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

In reality it didn't really hurt anything to have subject as a character, but you may encounter times when it is vital to change the data class, especially if numbers are read in as characters.

All the coercion functions have the form `as.[class]`, where you replace `[class]` with the class you would like

Logical Operators

In many instances you will need to use logical operators:

- `>`, `<`: greater than, less than
- `>=`, `<=`: greater than or equal to, less than or equal to
- `==`: equal to
- `!=`: not equal to

Logical expressions are evaluated as either TRUE or FALSE

```r
5 > 1
```

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

```r
5 < 1
```

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

We can also evaluate logical expressions across a vector

```r
1:10 > 5
```

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

Some more examples

```r
1 == 1
```

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

```r
2 == 1
```

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

```r
c(3, 9, 1) == 3
```

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

```r
c(2,25,-9,0) <= 0
```

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

```r
1:10 != 5
```

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

Logical expressions can be combined use the `&` (AND) and `|` (OR) operators

```r
TRUE & TRUE
```

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

```r
TRUE & FALSE
```

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

```
FALSE & FALSE
```

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

```
TRUE | TRUE
```

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

```
TRUE | FALSE
```

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

```
FALSE | FALSE
```

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

```
3 > 2 | 2 > 1
```

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

```
2 > 3 & 3 > 2
```

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

```
10 > 0 | 10 > 50
```

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

Creating new variables in your data frame

You just saw how you can overwrite one of the variables in your data frame using the `<-` operator. Now we will use that operator to create new variables.

Let's say we wanted each monkey's age in years rather than days.

```
lab_data$age_yrs <- lab_data$Age/365
```

When ever you do a mathematical operation on a data vector (i.e., `lab_data$Age`), the operation is performed on each item in the vector (i.e., each number is divided by 365). Here is a side by side comparison of the original age variable and the one we just created:

```
head(lab_data[ , c("Age", "age_yrs")])
```

```
##   Age   age_yrs
## 1 202 0.5534247
## 2 221 0.6054795
## 3 122 0.3342466
## 4 165 0.4520548
## 5 122 0.3342466
## 6 178 0.4876712
```

In the above example we created a new variable by basing it off of an existing variable in the data frame. But you don't have to base it off an existing variable, you can add any vector of values to the data frame as long it has the same number of rows as the data frame (with a few exceptions. For example, if we just wanted to add the numbers 1 through 26 as a column of data, we could do it like this:

```
lab_data$numbers <- 1:26
lab_data$numbers
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26
```

R also has a behavior called "recycling" — if the values you supply are less than the number of rows in the data frame, then it will repeat the values until the data frame is filled. This only works if the number of rows is divisible by number of items you supply

```
lab_data$seven <- 7 # This will repeat the value '7' 26 times
lab_data$one_to_thirteen <- 1:13 # This will repeat the values 1 through 13 twice

lab_data[,c("seven", "one_to_thirteen")]
```

```
##    seven one_to_thirteen
## 1      7               1
## 2      7               2
## 3      7               3
## 4      7               4
## 5      7               5
## 6      7               6
## 7      7               7
## 8      7               8
## 9      7               9
## 10     7              10
## 11     7              11
## 12     7              12
## 13     7              13
## 14     7               1
## 15     7               2
## 16     7               3
## 17     7               4
## 18     7               5
## 19     7               6
## 20     7               7
## 21     7               8
## 22     7               9
## 23     7              10
## 24     7              11
## 25     7              12
## 26     7              13
```

```
lab_data$no_recyling <- c(2, 4, "This won't work")
```

```
## Error in `$<-.data.frame`(`*tmp*`, no_recyling, value = c("2", "4", "This won't work": replacement ha
```

Descriptive Statistics in R

R has many functions for calculating central tendency.

```
mean(lab_data$Hair_Cortisol, na.rm = TRUE)
```

```
## [1] 78.7
```

```
median(lab_data$Hair_Cortisol)
```

```
## [1] 73.25
```

A note on missing values: In the lab data there are no missing values, but when you have data with missing values (labeled as `NA` in R) then for some of the functions above you will need to set the `na.rm'=` argument (stands for 'remove missing values') to `TRUE` (the default is `FALSE`)

```
# returns NA
mean(c(10, 10, NA, 4))
```

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

```
# returns a numeric
mean(c(10, 10, NA, 4), na.rm = TRUE)
```

```
## [1] 8
```

```
# To determine mode, you just have to look at the data. However, the table() function will give you a c
table(lab_data$Hair_Cortisol)
```

```
##
##  59.8  62.5  64.4  65.5  67.3  67.9  69.9  70.5  71.2  71.3  72.8  72.9  73.1
##     1     1     1     1     1     1     1     1     1     1     1     1     1
##  73.4  74.3  74.7  78.4    82  87.6    88  88.1    90  90.4 102.7 110.2 117.3
##     1     1     1     1     1     1     1     1     1     1     1     1     1
```

```
table(lab_data$Sex)
```

```
##
## Female   Male
##     12     14
```

To determine the standard deviation, use `sd()`

```
sd(lab_data$Hair_Cortisol)
```

```
## [1] 14.49381
```

To determine N, use `length()` on a vector, or `nrow()` on a data frame

```
length(lab_data$Hair_Cortisol)
```

```
## [1] 26
```

```
nrow(lab_data)
```

```
## [1] 26
```

To determine standard error of the mean, you will need to do some manual calculations. Recall that standard error is defined as the the standard deviation divided by the square root of N

$$\widehat{\mathrm{SE}}(\bar{X}) = \frac{s_X}{\sqrt{n}}$$

```
n <- length(lab_data$Hair_Cortisol)

sd_hc <-  sqrt(lab_data$Hair_Cortisol)

std_err_hc <- sd_hc / sqrt(n)

std_err_hc
```

```
##  [1] 1.646675 1.654830 1.860521 1.835546 1.839732 1.690471 1.587208 1.775907
##  [9] 1.573824 1.674469 1.864651 1.608870 1.550434 1.676764 1.616026 1.987461
## [17] 2.058752 1.639653 2.124038 1.840777 1.695015 1.673320 1.516575 1.736486
## [25] 1.655991 1.680201
```