Introduction to R

Tour of RStudio

Welcome to R and RStudio!

- R is the underlying statistical computing environment. Think of this like a car's engine.
- RStudio is the Integrated Development Environment that we use to interact with R, making writing and reading code a lot easier. Think of RStudio like the dashboard of the car.

RStudio Panes

- On the top left is the **script** window. This is where we are going to write all of our code and notes.
- On the lower left there's the **console** window. This is where R tells us what it thinks we told it and then the answer. This part is what R would look like (without RStudio)
- The top right has the **environment** and history tabs. The environment is a list of all objects that R knows and the history tab shows all the code that has been run.
- On the bottom right there's a window with lots of tabs. Files provides the file structure in the working directory. **Plots** is where your visualizations will appear. Packages shows all of the installed packages and one that are checked are opened. **Help** is where we will learn about functions when we need assistance and the Viewer is for viewing other kinds of output, like web content.

RStudio Global Options

- So that all of our set-ups look the same follow me to change a few settings. Go to Tools –> Global Options
  - In the Code menu check the box for "Soft-wrap R source files"
  - In the R Markdown menu UNcheck the box for "Show output inline for all RMarkdown documents"

Once these options are set, they will remain active for every R session you open on your local machine so you will not need to follow these steps again.

RMarkdown

Today we are coding in an **RMarkdown** document. This is how I would recommend that you code your own projects too. RMarkdown interweaves prose with code. Prose are written in plain text and R code is contained in gray "r code chunks".

To run code, place your cursor somewhere in the R chunk (between the lines with the backticks) and use `CTRL + ENTER` (also `CMD + ENTER` on mac)

When you run code, it is sent from the script to the console where it is evaluated. Then R returns "the answer"

The first thing we are going to do is read in data from a csv file. We are initially going to use the `read.csv` function which comes with your normal Base R installation. Later on, we will use a different version of this function. For now, just run the code and see what happens.

```
rodents <- read.csv("desert_rodents.csv")
```

Looking at our environment window, we can see that the `rodents` dataset has been created. By looking in our console at the bottom, we can see a record of the code that has been run.

The goal today will be to become familiar with coding concepts in R and develop proficiency reading, writing, and running code in the RStudio environment.

View a dataset

Let's find out what our dataset looks like by `View()`ing the dataset.

To `View()` the dataset in spreadsheet form, we can click on the dataset's name in the Environment tab. Notice that this action is accompanied by some code in the console telling us that we could also get there using code. Let's try it both ways

```
#Use the View function to investigate your dataset.
View(rodents)
```

In the r chunk above, I have written my first code comment. Anything after a `#` sign is a comment. Use them liberally to *comment your code* to explain to yourself and others what the code is doing and why you have chosen to do it this way.

- Commenting is helpful when you're testing things out during your analysis to 'turn off' parts of your script
- Comments are also a big part of making your work reproducible for others and for your future self when you open this script a few months from now and need to remember what you were doing
- Today I would like to challenge you to take notes directly in this document, *either as prose or as* `# comments` *in your r code chunks.*

The arrow <- creates R objects

The arrow operator is created with the less than sign followed directly by the dash.

We use it in R to create new objects.

So far in our coding today, we have used the arrow operator once already, resulting in 1 object in our environment:

1. We created our first object, `rodents`, when we read in our data from the csv file

Let's use the arrow to create a few more R objects. First, let's create an object containing one value.

```
object1 <- 55
```

Notice that when you run code that has an `<-` operator, the object is created in the environment. You will not see the value contained in the object in the console unless you ask R to print the object by calling its name:

```
object1
```

```
## [1] 55
```

```
# R is case sensitive! Watch your casing and spelling

# Object1
```

We can overwrite the value of object1 by re-assigning it

```
object1 <- 70

#then call its name to see the object
object1
```

```
## [1] 70
```

Now is a great time to save our script. Go to File –> Save or `CTRL+S` (pc) or `CMD+S` (mac).

Where is the file saving? Because I set up a project file at the beginning, your file is saving inside that folder. For your own work, I would suggest creating projects to make data/file management much easier.

Look at the environment. What does it tell you about object1?

Let's create some more R objects that are collections of several values. To accomplish this, we will use the function `c()`, which stands for concatenate or combine. Usually functions are named with a full word describing what they do but because combining items together is so common, this function gets a very short name.

```
object2 <- c(55, 60, 35, 70)
```

Check out the environment now. It worked! We created object2. This should look a lot like what we did with the weights in our powerpoint example. Here, y1 would be 55, y2 = 60 and so on.

Let's create another object containing a different type of data

```
object3 <- c("BIMS", "David", "8380")
```

Functions

A function is a verb; it tells R to do something. To call an R function, we call the name of the function followed directly by `()`. Recall our use of the `read.csv()` and `View()` functions. The items passed to the function inside the () are called *arguments*. Arguments change the way a function behaves

Let's `sum()` everything in object2

```r
sum(object2)
```

```
## [1] 220
```

Try the `mean()` function on your own for object2. We will learn more about means in the next lesson, but essentially you sum up all of the observations and then divide by the total number of observations.

```r
mean(object2)
```

```
## [1] 55
```

What happens if we try to `sum()` object3?

```r
# sum(object3)
```

This is why it is important to understand the type of data that you have.

What if we take the square root of object3? Or if we add object1 to object2

```r
sqrt(object2)
```

```
## [1] 7.416198 7.745967 5.916080 8.366600
```

```r
object1 + object2
```

```
## [1] 125 130 105 140
```

It worked! Most functions in R are **vectorized** meaning that they will work on a vector as well as a single value. This means that in R, we usually do not need to write loops like we would in other languages.

EXERCISE

Try the following functions on object2 and on object3. What do each of the below functions do? Optionally, call up the help menu for these functions to learn more.

1. class()
2. length()
3. summary()
4. str()

```r
class(object2)
```

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

```r
class(object3)
```

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

```
length(object2)
```

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

```
length(object3)
```

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

```
summary(object2)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    35.0    50.0    57.5    55.0    62.5    70.0
```

```
summary(object3)
```

```
##    Length     Class      Mode
##         3 character character
```

```
str(object2)
```

```
##  num [1:4] 55 60 35 70
```

```
str(object3)
```

```
##  chr [1:3] "BIMS" "David" "8380"
```

Functions to inspect dataframes

R has a few different types of objects. We already saw some vectors (one dimensional collection of items) before when we created object2 and object3.

R's dataframes store two dimensional, tabular, heterogeneous data. Two-dimensional and tabular meaning a table of rows and columns that form the 2 dimensions. Heterogeneous meaning that each column can contain a different type of data (i.e., one column Age is numeric while Gender is a character).

A dataset is considered "tidy" when each variable forms a column, each observation forms a row, and each cell only contains one piece of data. This means that the entries within a column should all be the same type as each other.

Let's remind ourselves how to look at the whole dataset. Does anyone remember from before, how do we see the whole dataset in the spreadsheet viewer?

```
View(rodents)
```

We can also directly call the `dim()` function to see the dimensions of the `rodents` dataset.

```
dim(rodents) #rows then columns
```

```
## [1] 21 11
```

We can ask for the number of rows and the number of columns

```
nrow(rodents)
```

```
## [1] 21
```

```
ncol(rodents)
```

```
## [1] 11
```

To see all the column headings we can call the function `names()`

```
names(rodents)
```

```
##  [1] "species"        "scientificname" "commonname"     "granivore"
##  [5] "minhfl"         "meanhfl"        "maxhfl"         "minwgt"
##  [9] "meanwgt"        "maxwgt"         "juvwgt"
```

And probably the two you'll use the most to inspect data frames, because they are the most descriptive, are `summary()` and `str()`, both of which we used above to inspect vectors

```
summary(rodents)
```

```
##    species          scientificname      commonname          granivore
##  Length:21          Length:21          Length:21          Min.   :0.0000
##  Class :character   Class :character   Class :character   1st Qu.:0.0000
##  Mode  :character   Mode  :character   Mode  :character   Median :1.0000
##                                                           Mean   :0.7143
##                                                           3rd Qu.:1.0000
##                                                           Max.   :1.0000
##
##      minhfl         meanhfl          maxhfl          minwgt
##  Min.   : 6.0   Min.   :13.25   Min.   :15.00   Min.   : 4.00
##  1st Qu.:11.0   1st Qu.:19.95   1st Qu.:23.00   1st Qu.: 6.00
##  Median :15.0   Median :21.48   Median :31.00   Median : 9.00
##  Mean   :15.1   Mean   :24.17   Mean   :33.57   Mean   :10.24
##  3rd Qu.:19.0   3rd Qu.:26.03   3rd Qu.:39.00   3rd Qu.:13.00
##  Max.   :39.0   Max.   :49.93   Max.   :64.00   Max.   :30.00
##
##     meanwgt           maxwgt           juvwgt
##  Min.   :  7.90   Min.   : 16.00   Min.   : 5.867
##  1st Qu.: 17.47   1st Qu.: 28.00   1st Qu.:13.203
##  Median : 24.06   Median : 48.00   Median :18.714
##  Mean   : 40.92   Mean   : 79.29   Mean   :26.336
##  3rd Qu.: 49.04   3rd Qu.: 85.00   3rd Qu.:28.701
##  Max.   :162.53   Max.   :280.00   Max.   :83.752
##                                    NA's   :3
```

```
str(rodents)
```

```
## 'data.frame':    21 obs. of  11 variables:
##  $ species       : chr  "BA" "PB" "PH" "PI" ...
##  $ scientificname: chr  "Baiomys taylori" "Chaetodipus baileyi" "Chaetodipus hispidus" "Chaetodipus
##  $ commonname    : chr  "Northern pygmy mouse" "Bailey's pocket mouse" "Hispid pocket mouse" "Rock po
##  $ granivore     : int  1 1 1 1 1 1 1 1 0 0 ...
##  $ minhfl        : int  6 16 21 18 11 21 15 39 21 12 ...
##  $ meanhfl       : num  13.3 26 25.1 22 21.5 ...
##  $ maxhfl        : int  15 47 28 24 27 50 64 58 42 39 ...
##  $ minwgt        : int  6 10 18 10 4 13 12 12 30 7 ...
##  $ meanwgt       : num  9.45 31.87 30.72 17.47 17.62 ...
##  $ maxwgt        : int  18 79 48 28 42 66 85 190 280 56 ...
##  $ juvwgt        : num  NA 19 24 10 11.7 ...
```

Accessing variables from a dataframes

You might have noticed a `$` in front of the variable names in the `str()` output. That symbol is how we access individual variables, or columns, from a dataframe

The syntax we want is dataframe$columnname Let's look at the `title` column

```
rodents$commonname
```

That function calls the whole column, which is 21 observations long. Usually printing out a long vector or column to the console is not useful.

`head()` is a function allowing us to look at just the first 6 entries

```
head(rodents$commonname)
```

```
## [1] "Northern pygmy mouse"   "Bailey's pocket mouse"  "Hispid pocket mouse"
## [4] "Rock pocket mouse"      "Desert pocket mouse"    "Merriam's kangaroo rat"
```

What if we want to see the first 10 values?

Let's see if we can find out by calling help on the `head()` function

```
?head
```

```
## starting httpd help server ... done
```

The help menu tells us what `head()` does and it also specifies the other arguments that we could input to the `head()` function in the Arguments section. This is always a good section to check out. Remember that an argument is an option we specify to a function to change how the function operates.

Let's try adding the `n =` argument to `head()`

```
head(rodents$commonname, n = 10)
```

```
##  [1] "Northern pygmy mouse"       "Bailey's pocket mouse"
##  [3] "Hispid pocket mouse"        "Rock pocket mouse"
##  [5] "Desert pocket mouse"        "Merriam's kangaroo rat"
##  [7] "Ord's kangaroo rat"         "Banner-tailed kangaroo rat"
##  [9] "White-throated woodrat"     "Northern Grasshopper Mouse"
```

Although we can specify the head function without naming the arguments, it is good practice to label the arguments to clarify what the code is doing. However, it is conventional to skip labeling the first argument, x, since its label is easily assumed.

You may have read in the help menu that `head()` has a companion function `tail()` that shows the last n rows

```
tail(rodents$commonname, n = 10)
```

```
##  [1] "Silky pocket mouse"       "Cactus mouse"
##  [3] "White-footed mouse"       "Deer Mouse"
##  [5] "Fulvous harvest mouse"    "Western harvest mouse"
##  [7] "Plains harvest mouse"     "Tawny-bellied cotton rat"
##  [9] "Hispid cotton rat"        "Yellow nosed cotton rat"
```

Let's calculate some descriptive statistics for the temperature at each sighting using the `temperature` column/variable. Again, we will look at the equations for these functions later on.

```
mean(rodents$meanwgt)
```

```
## [1] 40.92359
```

```
sd(rodents$meanwgt)
```

```
## [1] 39.82418
```

```
median(rodents$meanwgt)
```

```
## [1] 24.056
```

```
IQR(rodents$meanwgt)
```

```
## [1] 31.57285
```

```
range(rodents$meanwgt)
```

```
## [1]   7.899749 162.534774
```

For variables where there are missings, we will need to include an argument that removes the missings. Try to calculate the mean of the `juvwgt` variable.

```r
mean(rodents$juvwgt)
```

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

```r
?mean
```

Looking at the arguments section tells me that the argument I need to include is `na.rm = TRUE`

```r
mean(rodents$juvwgt, na.rm = TRUE)
```

```
## [1] 26.33586
```

EXERCISE

1. What's the standard deviation for minimum weight (`minwgt`)?
2. Use the `sum()` function and the `granivore` variable to count how many rodents are granivores (feed on grain)
3. What's the maximum mean hindfoot length (`meanhfl`) for a rodents species in this dataset (hint: `max()`)?

```r
sd(rodents$minwgt, na.rm = TRUE)
```

```
## [1] 6.147396
```

```r
table(rodents$granivore)
```

```
##
##  0  1
##  6 15
```

```r
max(rodents$meanhfl)
```

```
## [1] 49.92785
```

Additional Functionality -> Installing and Loading Packages.

The next two sessions of this class will show how to prepare data in a dataset and how to graph/visualize your data.

In order to use additional functionality in R, we bring in packages. The `install.packages()` installs the package to your local machine, while the `library()` command loads the functions from that package into the R environment so that we can use them. You need to run the `library` function everytime you open a new script/markdown file, but you do not need to run the `install.packages()` function everytime.

```r
#install.packages("tidyverse")
library(tidyverse)
```

In this case, the code opened a library containing R functions we want to use. You can think of libraries like apps on your phone. We have now opened the app so we can use it. The output in the console specifies which libraries we have loaded. We can see based on the output from the `library(tidyverse)` line that the tidyverse is actually a megapackage, containing 8 packages. All of these packages share a similar syntax in an attempt to simplify coding and readability for R users. Aside from the core tidyverse packages, there are around 10 other packages

Below, we are using the `read_csv` function rather than the `read.csv` function. The `read_csv` function reads in your dataframe as a tibble, which is essentially a dataframe with some added functionality and aesthetic flourishes.

The `read_csv` function is only available if you are able to load up the `readr` package which is a part of the `tidyverse`.

```
new_rodents <- read_csv("desert_rodents.csv")
```

```
## Rows: 21 Columns: 11
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): species, scientificname, commonname
## dbl (8): granivore, minhfl, meanhfl, maxhfl, minwgt, meanwgt, maxwgt, juvwgt
##
## 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.
```

Further Resources

Learning more R

- R4DS: R 4 Data Science - a free e-book with excellent, engaging exercises
- RStudio Primers Engaging online lessons to teach you introductory R skills
- RStudio's cheat sheets
- Jenny Bryan's Stat 545 "Data wrangling, exploration, and analysis with R" course material: An excellent resource for learning R, dplyr, and ggplot2

Troubleshooting coding

- Read package vignettes. For example, see the introduction to dplyr vignette.
- Google it!: Try Googling generalized versions of any error messages you get. That is, remove text that is specific to your problem (names of variables, paths, datasets, etc.). You'd be surprised how many other people have probably had the same problem and solved it.
- Stack Overflow: There are over 100,000 questions tagged with "R" on SO. Here are the most popular ones, ranked by vote. Always search before asking, and make a reproducible example if you want to get useful advice. This is a minimal example that allows others who are trying to help you to see the error themselves.