STAT 213 HW0a

R/RStudio Intro / Basic Descriptive Stats

Last Revised February 5, 2018

1 Starting R/RStudio

There are two ways you can run the software we will be using for labs, R and RStudio.

Option 1 is to log in to the Oberlin RStudio server from a web browser (lab or personal computer work equally well). Option 2 is to install the software on your own computer. The advantage

Advantages of the server approach: you can get to your account, and your files, from anywhere, you don't need to bring your laptop to lab, and you don't have to install anything.

Disadvantages of the server approach: you have to upload and download files to the server if you create them on your computer and want to use them in RStudio, or if you want to do something with a file that you created in RStudio, there may be occasional server outages during which you can't work on your homework, or slow downs when a lot of people are using it at once.

Logging in to the RStudio Server

This section is for those who want to access the software from a web browser.

1. In your web browser, visit `rstudio.oberlin.edu`.
2. If you filled out the background survey, your username is your Obie ID (the short form of your email, not including the @oberlin.edu), and the initial pass-

word is the same as your username. Let me know if you don't have an account yet.

3. Before moving on, you should change your password. From the Tools menu in RStudio (not your browser), select Shell.... A window will open with a blinking cursor. Type `passwd`. You will be prompted to enter your old password, followed by a new password twice. As you type you will not see anything appear, nor will the cursor move, but the computer is registering your keystrokes. When done you can close this window.

Installing R and RStudio Locally

This section is only needed if you are working on your own laptop and you want to have a local install of the software. If you just want to work on the server, you can skip this part.

Download and Install R

1. Visit `cran.r-project.org` and click the link at the top of the page for your operating system. Most likely you want the first available link on the resulting page.
2. *Mac*: Assuming you have a recent version of OS X, click on the first available link, download the `.pkg` file, and install it as you would normally install an application on OS X. *Windows:* Select "base", and then "Download R 3.2.2 for Windows". Open the executable and follow the instructions of the installer.

Download and Install RStudio

1. Visit `www.rstudio.com`, and click the button labeled "Download RStudio" in the top panel (the panel will cycle through a few screens — the one we want is on the first one).
2. Click the link for "RStudio Desktop", and then "Download RStudio Desktop" under "Open Source Edition".
3. Under "Installers", select the link for your operating system to download the installer.

4. Open the installer, and follow the instructions to install RStudio.

2 Navigating the RStudio Environment

The RStudio environment consists of **panes**, each of which may have several tabs (as in a web browser).

When you first open RStudio, the left-hand pane (by default) displays the **Console**. Here, you can enter commands one at a time to see what they do.

Enter some simple arithmetic expressions at the console and verify that the output is as expected. (Note that the asterisk represents multiplicationn and the slash is for division.)

```
2 + 5
13 * 17
17.04 / 2.8
sqrt(49)
```

> Useful tip: When your cursor is at the console, you can use the up and down arrows to cycle through recently typed commands.

In the lower right is a pane with various tabs to do things like keep track of your files, view graphs and plots that you create, get help messages, and see what **packages** are installed. Since R is open and free, anyone who wants to can create a module of code to extend or streamline functionality, and that others can download and use.

Installing and Using R Packages In each session, any time you want to use commands that are part of a package that is not part of core R, you need to tell R to look in that package for names of functions and datasets, using the `library()` function. One such package we will use a lot is called `mosaic`. Try typing the following:

```
library("mosaic")
```

Note the double quotes. (In this particular case the command would work without the quotes, but it is good practice to include them, as often times leaving out quotes will produce an error.)

2.1 Package Installation (SKIP IF ON THE SERVER)

If you just installed R, you probably got an error message, to the effect that the `mosaic` package does not exist! It is already installed on the server, but you will need to install it on your own machine (you should only need to do this once). Type

```
install.packages("mosaic")
```

You may get an error message that some other package (which `mosaic` depends on) is not found. In this case modify the above line as follows:

```
install.packages("mosaic", dependencies = TRUE)
```

The `dependencies = TRUE` bit tells R to automatically install any other packages that are required.

While you're at it, also install some other packages that contain datasets we will use (again, you can skip this step if you are on the server).

```
install.packages("Stat2Data", dependencies = TRUE)
install.packages("Lock5Data", dependencies = TRUE)
```

You only need to do this one time — even if you quit and restart RStudio, these packages will be installed and available.

3 Data Frames

Definitions The people or entities corresponding to a single data point are called **cases**. The characteristics that are being measured (like height, age, blood pressure) are called **variables**. For example, if we created a dataset recording the major, year, and age of the students in this class, the cases would be the individual students, and the variables would be major, year, and age.

Datasets are represented in R as tables called **data frame**s, where each row represents a case, and each column represents a variable. We can see a list of datasets available in a particular package (say, `Stat2Data`) by typing the following:

```
data(package = "Stat2Data")
```

Note the quotes around the package name, but *not* around the word package. We will see why this is the case soon. Note also that everything in R is case-sensitive. If I don't capitalize names exactly as written, I will get an error. For example,

```
data(package = "Stat2data")

## Error in find.package(package, lib.loc, verbose = verbose):  there is
no package called 'Stat2data'
```

If we are going to be working with a particular package a lot, we can "load it into memory" (for the duration of our R session) by doing

```
library("Stat2Data")
```

Once we have loaded the package, we can load an individual dataset from this package, such as `Pollster08`, by typing

```
data("Pollster08")
```

We can look at the **code book** (the description of what the data is, how it was collected, and what each variable means) by typing

```
?Pollster08
```

This will only work if we have first made the package visible using the `library()` command.

Exercise 1 What are the first three variables in the `Pollster08` dataset? What are the cases? For now, jot your answers to these exercises on a piece of paper, or wherever you want. At the end you will learn how to record code in a file called an "R Script", where you will copy your answers to turn in.

We can look at the first few rows of the dataset with the `head()` function:

```
head(Pollster08, n = 4)
```

The `n = 4` indicates that we want to see the first four rows. We can leave this part out and just type `head(Pollster08)`, which will show us the first six rows by default. We can also examine a dataset in RStudio's spreadsheet-like viewer by double-clicking on the name of the dataset in the Environment pane in the upper right.

We can ask R to report the number of rows (cases) and the number of columns (variables) in the data using `nrow()` and `ncol()`:

```
nrow(Pollster08)
ncol(Pollster08)
```

Exercise 2 Answer the following questions about some datasets.

(a) Write a brief summary of what the dataset `HoneybeeCircuits` from the `Lock5Data` package is about. (Hint: Look at the code book)

(b) How many cases are there in this dataset? (Bonus Brownie points if you use an R command to get this value)

(c) What's the third variable in the `EmployedACS` dataset from the `Lock5Data` package?

4 Functions, Arguments, and Commands

Most of what we do in R consists of applying **functions** to data objects, specifying some options for the function, which are called **arguments**. Together, the application of a function, together with its arguments, is called a **command**.

A useful analogy is that commands are like sentences, where the function is the verb, and the arguments (one of which usually specifies the data object) are the nouns. There is often a special argument that comes first. This is like the direct object of the command.

For example, in the English command, "Draw a picture for me with some paint", the verb "draw" acts like the function (what is the listener supposed to do?); the noun "picture" is the direct object (draw what?), and "me" and "paint" are extra (in this case, optional) details, that we might call the "recipient" and the "instrument". In the grammar of R, I could write this sentence like:

```
### Note: this is not real R code
draw("picture", recipient = "me", material = "paint")
```

We are applying the function `draw()` to the object "picture", and adding some additional detail about the recipient and material. Here the function is called `draw`,

and we have a main argument with the value `"picture"`, and additional arguments `recipient` and `material` with the values `"me"`, and `"paint"`, respectively.

Technically speaking, "picture" is the value of an argument too; we might have written

```
### Note: this is not real R code
draw(object = "picture", recipient = "me", material = "paint")
```

However, in practice, there is often a required first "main" argument whose name is left out of the command.

In R, arguments always go inside parentheses, and are separated by commas when there is more than one. For arguments whose names are explicitly given, the name goes to the left of the `=` sign, and the value goes to the right.

When above we said

```
head(Pollster08, n = 4)
```

We are applying the function `head()` function to the dataset `Pollster08`, and giving an extra detail, `n = 4` that modifies what happens. Here, the function is called `head()`, the main argument's name is left out, but has the value `Pollster08`, and the second argument has the name `n` and the value `4`.

Sometimes the value will have quotes around it. The name will not. More on this distinction shortly.

Exercise 3 Inspect the following command that was used above:

```
head(Pollster08, n = 4)
```

Identify all of the function names, argument names, and argument values. Note that sometimes argument values are supplied *without* an argument name: R can assign values to names according to the order they're given in. However, the reverse is never true: if you give an argument name you need to supply a value.

5 Objects and Variables

We already saw one kind of object: a data frame. Other kinds of objects include **text strings**, which are written with quotes around them (these are often used as argument values), **literals**, such as numbers (like the 4 above — no quotes) and the special values `TRUE` and `FALSE` (note the all caps). There are also objects called **vectors**, which are like lists that can either contain text strings or numbers. We haven't seen any of these yet, but each variable (column) in a data frame is represented as a vector.

When we want to use an object a lot (such as a numeric value, like a mean, from some statistical computation), it is helpful to give it a name so we can refer to it by what it represents, instead of by its values. Data frame objects almost always have names, so that we can refer to them.

We can give a name to an object using an expression of the form `name <- value`. This process is called **assignment**, and the named thing is called a **variable** (which means something different than a variable in statistics). For example:

```
my.name <- "Colin Dawson"
my.age <- 35
```

You can read the `<-` symbol as "gets", as in "(The name) `my.name` gets (the value) `''Colin Dawson''` ". Notice that there is just one hyphen. A common error is to add an extra hyphen to the arrow, which R will misinterpret as a minus sign. Again, the periods in the variable names are just part of the name.

It is also legal to use underscores and digits in variable names, but none of these can be used at the beginning of a name.

We can also store the result of a command in a named variable. A simple example is the following:

```
fred <- sqrt(25)
```

Now if I type the name of the new variable at the console, `R` will print out its contents:

```
fred
```

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

The 1 in brackets is there to indicate that the next value shown is the first entry in

the variable called `fred` (Note that if you try to access the variable `Fred`, you will get an error, because you defined it with a lower case "f"). In this case the variable has only one entry, but sometimes we will hold lists of data or other values in a variable.

We can also use variables as the values of arguments, such as in:

```
a.squared <- 3^2
b.squared <- 4^2
a.squared.plus.b.squared <- a.squared + b.squared
```

Notice that every time we define a variable, it appears in the upper right RStudio pane, in the Environment tab. This shows us everything we've defined.

If you want to read in a dataset from a file, you need to give it a name within R. There is a dataset on my website involving measurements of depression in 18 people. Type the following to get it and save it to a data frame called `Depression`.

```
Depression <- read.file("http://colindawson.net/data/depression.csv")
```

Note that file names are always in quotes because they refer to a location outside the R environment itself, not to a variable.

You may get an error message at this point to the effect that R can't find the function `read.file()`. This is because it lives in the `mosaic` package, and we need to activate it (you may have done this already, in which case you will not get an error message). Do

```
library("mosaic")
```

and then run the `read.file()` line again.

Now that the `Depression` data frame is defined, take a look at the first five rows with the `head()` function:

```
head(Depression, n = 5)
```

Since this dataset was read in from a text file, R does not have a code book associated with it, so doing `?Depression` will result in an error.

Exercise 4 Create a named variable that stores the number of cases in the `Depression` dataset. The right-hand side of your assignment operation should be a command that returns the number of cases, not just the number (as in the `fred` example above, where we used the `sqrt()` function to compute the value we wanted).

Exercise 5 Each of the following commands has an error in it. What is it?

(a) `ralph <- sqrt 10`

(b) `ralph2 <-- "Hello to you!`

(c) `3ralph <- "Hello to you!"`

(d) `ralph4 <- "Hello to you`

6 R Scripts for Reproducible Research

Typing directly into the console is useful for experimenting. But when it comes time to do finished work, you want to be able to reproduce what you did. Instead of typing directly into the console, you can record lines of code in a file (called a "script"), and have R execute them back to back.

Creating a Script in RStudio Under the File menu, select New File, and choose R Script. used to create new files. Click here and select "R Script".

The left-hand pane will divide in two, with the console at the bottom, and the top a blank page. You can enter code on this page, line by line, and click "Run" to execute the current line, or the highlighted region (you can also press Control-Enter (or Command-Enter on a Mac) to run the current line). Alternatively, you can run the whole file at once by clicking "Source".

Any code preceded on the same line by a pound sign – or "hashtag" if you prefer — (#) is interpreted as a "comment", and will be ignored when running that line or the whole script. For example:

```
### This part of the code is an illustration of comments

## This line is a comment, so it doesn't have to be valid R code
library("Stat2Data") # Making Stat2Data visible to commands
data("BritishUnions") #Loading a dataset
```

Note that comments can either be on a line by themselves, or at the end of a line. A single pound sign is enough to identify something as a comment, but by convention, in-line comments have a single pound, whole line comments have two pound signs, and comments that pertain to a whole section of code have three or more.

To save your work on a script, select Save or Save As... from the File menu and give your script a name. The convention is to give R scripts the extension `.R`, as in `MyFirstScript.R`.

Exporting a Script from the Server If you are working on the server, you will often need to move data files and script files between your own computer and the server (for example, to turn in an R script as homework).

Once you have saved your script:

1. Select the Files tab in the lower right pane of RStudio.
2. Find the file you just saved and check the box next to the file name.
3. Click the More button (with a gear on it) between the list of tabs and the list of file names, and select Export. (Note that you can export multiple files at once this way as well.)
4. A dialogue box will open allowing you to rename the file if you choose. Click Download. Your browser will then generate a download dialogue to save the file to your computer as you would normally download a file from the web, saving to your Downloads folder, or whatever location you have set your browser to use.

You can also upload files from your computer to the server using the Upload button in the same row as the More button. You will need to do this if you create your own data file in Excel, for example (as you are likely to do for the project).

Exercise 6 Type your answers to Exercises 1-5 in a script, with clearly labeled headings for each exercise. Include any code you needed to run as code, and any verbal responses (and headings) as comments.

Include any previous code needed to make your code run. To verify that your script is self-contained, select "Restart R", followed by "Clear Workspace" from the Session menu. This will wipe away any packages you have made visible, and any variables you have defined. Then, run the script. If you get errors that you did not get before clearing everything, you are missing something.

Add to this script as you continue with the lab.

7 Describing and Visualizing Data

In this section we will look at how to create some basic visualizations and get some basic summary statistics in R.

7.1 Quantitative vs. Categorical Variables

We need a couple of definitions first.

Definitions A **quantitative variable** corresponds to a measurement of a case on some scale. Arithmetic makes sense for a quantitative variables. For example, height, or GPA would be quantitative: you can take a sum or difference between two heights or GPAs.

A **categorical variable** is a "qualitative" variable that divides cases into groups. For example, major, or favorite color would be categorical. A variable like T number is also categorical even though it is represented using a number; arithmetic doesn't make sense.

A categorical variable with only two possible values (such as 'yes' or 'no') is called **binary**.

Exercise 7 Examine the documentation for the `EmployedACS` dataset from the `Lock5Data` package, including the description of the values of each variable. Which ones are quantitative? Which are categorical? Which are binary?

7.2 Visualizing a Quantitative Variable

For a quantitative variable like `HoursWk` (Hours worked per week), we can get a general sense of what values are in the data using a **histogram**.

```
library("mosaic")    # Needed for the histogram() function
library("Lock5Data") # Needed for the EmployedACS data
data("EmployedACS")  # Loading the dataset
histogram(~HoursWk, data = EmployedACS, type = "count")
```

The first argument here is a "formula", with the name of the variable we are interested in plotting appearing on the right hand side of the tilde ($\sim$) symbol. We will see this format appear repeatedly.

We will also see the `data=` argument appear over and over again. This indicates what data frame the variable we are interested in is in.

Finally, the `type=` argument controls what we plot on the y-axis: here, the number of observations in each bin, or set of values of the quantitative variable.

Exercise 8 Before moving on, estimate the mean and median number of hours worked by looking at the histogram.

We can also create a **box-and-whisker plot** for a quantitative variable, which depicts the median, the **first and third quartile** (the value with 25% of the cases below it and the value with 25% of the cases above it) as the edges of the box, and draws "whiskers" which are 1.5 times the width of the box. Points beyond the whiskers are plotted individually.

```
bwplot(~HoursWk, data = EmployedACS)
```

Exercise 9 Estimate the first and third quartile of number of hours worked per week from the box-and-whisker plot.

Optional: Controlling Color The default plot colors are kind of ugly, in my opinion. You can control the color of the dots or bars, etc. using the `col=` argument. For example:

```
histogram(~HoursWk, data =  EmployedACS, width = 5, col = "forestgreen")
```

Type `colors()` at the console to see a (long) list of available colors. Alternatively you can use a predefined color palette and then refer to colors by number. For example, try this:

```
library("RColorBrewer")  # a package with some decent palettes
## Sets the palette to the first 8 colors in "Set1"
## Type ?brewer.pal to see other palette options
palette(brewer.pal(n = 8, name = "Set1"))
```

Now if we give colors by number it will use the palette:

```
histogram(~HoursWk, data = EmployedACS, width = 5, col = 3)
```

7.3 Summarizing a Quantitative Variable

We can get a set of useful summary statistics for our `HoursWk` variable using the `favstats()` function. The syntax is the same as for `histogram()`, in that we have a formula and a `data=` argument.

```
favstats(~HoursWk, data = EmployedACS)
```

Find the mean and median in the results. Were your estimates close?

We can get individual summary statistics using the same format. For example,

```
mean(~HoursWk, data = EmployedACS)
median(~HoursWk, data = EmployedACS)
sd(~HoursWk, data = EmployedACS)      # Standard Deviation
```

```
var(~HoursWk, data = EmployedACS)      # Variance
```

7.4 Summarizing a Categorical Variable

Mean, variance, etc. do not make sense for a categorical variable, since we cannot add the values together. Instead, we can summarize the variable by counting how often each value occurs:

```
tally(~Race, data = EmployedACS)
```

Or, instead of counts, we can get proportions in each category as follows:

```
tally(~Race, data = EmployedACS, format = "proportion")
```

Exercise 10 What is the total percentage of the sample that is non-white?

Optional R Tip Sometimes when using commands like `tally()` that return multiple values at once, we want to be able to access specific entries as part of another calculation. We can do this by **indexing** into the variable (which is represented as an "array", or a list of values). First, let's store the results of `tally()` as a variable, so we can work with them:

```
race.stats <- tally(~Race, data = EmployedACS, format = "proportion")
```

We can access individual entries of this table (array) either by position or by label using square brackets after the name of the table. For example, if I want the first entry, I can do

```
race.stats[1]
```

(Coders: note that `R` uses indices that start with 1, not 0 like Python and many other languages.)

I can also access entries by their name, for example:

```
race.stats["asian"]
```

I can even access multiple entries at once. If I want consecutive positions, I can use a colon:

```
race.stats[1:3]
```

If I want arbitrary positions, or a set of labels, I can use the `c()` function to "combine" a list of entries separated by commas. For example:

```
race.stats[c(1,2,3)]
race.stats[c("asian", "black", "other")]
```

I can then use these values in arithmetic expressions, or pass them to functions such as `mean()`, `sum()`, etc.

7.5 Visualizing a Categorical Variable

We can plot counts or proportions using a **bar graph**.

```
bargraph(~Race, data = EmployedACS)
```

It is good practice to tell R that a variable is categorical by using the `factor()` function. Sometimes categorical variables are represented using numbers, and R will not know to treat it as categorical unless you tell it.

Or, with proportions:

```
bargraph(~Race, data = EmployedACS, type = "proportion")
```

(Note that the argument name congrolling the y-axis has a different name for `bargraph` than it does for `tally`.)

Exercise 11 Try to create a bargraph showing the proportions of cases that do and do not have health insurance. Is there anything unsatisfying about the graph if you were going to put it in an article or paper? Modify your graph by adding a label for the x-axis using the `xlab=` argument. Put the desired label in quotes as the argument value so the reader knows what the bars represent.

8 Summarizing and visualizing subsets of the data

8.1 Filtering and Piping

Sometimes we will want to analyze or visualize only some of the cases in the data. For example, suppose we want to look at a histogram of weekly hours worked only for female workers.

We can produce a "reduced" data frame with the `filter()` function, as in the following:

```
filter(EmployedACS, Sex == 0)
```

The first argument is the name of the full dataset, and the second is a "filter" condition. Note the double equals sign. (If we have a quantitative filtering variable, we can also define filters using `>` or `<` for greater-than or less-than, or `>=` or `<=` for greater-than-or-equal-to and less-than-or-equal-to).

The above command on its own will just print out the filtered dataset, just as if we had typed `EmployedACS` at the console. What we really want is to use the result of this function as the value of the `data=` argument of whatever plotting or summary function we want.

Essentially, we want to "chain" or "nest" the two commands.

We have three options, which are all equivalent; you should use whichever one you find most natural.

First, we can directly nest the two commands, substituting the whole `filter()` command directly into the `data=` argument for `histogram()`, say:

```
histogram(~HoursWk, data = filter(EmployedACS, Sex == 0))
```

This can get messy quickly, however, so instead we can "save" the filtered data as a new object and then use that in `histogram()`.

```
EmployedACS.Males <- filter(EmployedACS, Sex == 0)
histogram(~HoursWk, data = EmployedACS.Males)
```

Finally, we can "chain" the commands together using the 'pipe' operator, which is written like `%>%`. This works like this:

```
filter(EmployedACS, Sex == 0) %>%
    histogram(~HoursWk, data = .)
```

where the . represents the output of the "pipe".

The same structure works if we want a mean or variance or whatever else from the filtered data.

8.2 Plotting and Summarizing by Group

If we want to create a plot or compute summary statistics for one variable for *every* different value of another, we can modify our formula expression so that the thing we are summarizing is on the left of the ∼, and the grouping variable is on the right:

```
bwplot(HoursWk ~ factor(Sex), data = EmployedACS)
```

The `factor()` function is used to tell R that the grouping variable is categorical, even if it is represented in the data using a number (as it is here, as 1 or 0).

Exercise 12 Compute the mean income for workers with and without health insurance. Do this in two ways: first using `filter()` to select each subset, and second using a grouping formula.