Tutorial 1: Welcome to R

Leah Brooks

January 14, 2019

Welcome! This tutorial assumes that you've already done everything in "Get R Ready" tutorial. If this is not the case, do that first.

Today we are working on basic data management commands and becoming familiar with the R interface. Sadly, no graphs today. But today should show you the power of statistical software relative to Excel, and prepare you to make graphics next class.

A. Open RStudio

RStudio is an interface for using R.

1. Open up RStudio.
2. What you need to know about what you see
    a. Console: This is the window where you input R commands. You can input them one by one, or you can write a R program, we will do.
    b. Terminal: This window is a terminal window. On a Mac, it should be a unix interface; on a PC it is a DOS interface. I don't anticipate using this window, but you could use it to find the location of files and execute system commands.
    c. Environment: Ignore for now!
    d. History: This reports a history of your R commands. Hopefully you won't need to refer to this window since you won't be programming interactively.
    e. Connections: Ignore for now
    f. Files: Ignore for now
    g. Plots: Your plots will appear here – if we do well, we'll see one this class.
    h. Packages: This lists packages you have installed (they sit on your hard drive). You should see the package "rmarkdown" in this list.
    i. Help: Help for commands. Alternatively, you can type `help("command")` at the console prompt.
    j. Viewer: Ignore for now.

B. Hello world program

The very first program people usually write in any language is a very simple one that prints "hello world" to the screen.

You should always write R code (or any code) in a program. Never program interactively! (This means typing things in one command at a time at the prompt.) Interactive programming is a way to lose track of what you're doing and make unreplicable work.

To write a R program, open RStudio, and choose File -> New File -> RScript.

You should see a new window open up.

In this new window, begin by writing a comment that says what your program does and when you worked on it. This is good practice, even though it may seem silly for this program.

When you write these comments, use a "#" sign in front of each line, so R will know that this is a comment and not code to evaluate.

My comments look like this

```
#
# this is my program to say "hello world"
#
# hiworld.r
#
# january 7, 2019

###### A. print hello world
```

After your comments, write the R command

```
print("Hello World!")
```

Save this file with a name you'll remember in a location you'll remember. Mine is saved as H:/pppa_data_viz/2018/assignments/lecture01_helloworld.R (If you type the file name, R will automatically add the .R extension to the name). This extension, .R,means it is an R program. Without the proper extension, the program will not work.

Now we want to run this program.

There are two ways to run this file (also known as a script)

1. While in the editor window, go to the Code menu -> Run Region -> Run All (or click the "run" bottom at the top of the window).
2. At the prompt in the console window, type the full name of your file and use the option "echo=TRUE" so that R will show all the individual commands that you're using.

```
source('H:/pppa_data_viz/2018/assignments/lecture01_helloworld.R', echo=TRUE)
```

Either way, you should see something like the below in the Console window:

```
print("Hello World!")
```

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

That's success! Now you're ready to work on a more complicated program with real data.

C. Loading Data

We begin by loading data into a dataframe. A dataframe is basic data structure we'll use most of the time in this course. Think of it like an excel table with rows and columns.

Start a new R program that you'll use for this class and for your homework. For the homework, you should turn in annotated code (that's a R Script with comments) that includes the class commands and the homework.

Use one .R file for the rest of this tutorial (and make one for each subsequent tutorial) and keep building on it. Do not program interactively (enter commands at the prompt; good for testing, but bad for real work)! This is a way to (a) not remember or understand what you did and (b) get the ordering logic in a program wrong.

We'll begin by loading a dataset where each observation is a county in the Washington metropolitan area (or in Virginia, independent city) in a year, 1910 to 2010. These data come from the Decennial Census, and you can find source info in this paper of mine.

In R, we usually read .csv (comma separated values) files, and I have prepared files in this format for today.

Download the data for this class from here. We begin by loading the file called "was_msas_1910_2010_20190107.csv". If you're curious what a .csv file looks like, you can open it in Excel.

Here's a simple example of a .csv file:

var1, var2, var3

a, b, c

d, e, f

g, h, i

where each row in this example is an observation. Each observation has three variables: var1, var2, and var3.

Variables (var1, var2, var3) are separated by commas.

Note where you save the data! You need to know the full path of the directory where it is saved, and the name you saved it under.

To load the file, we use the read.csv command. The input to this command is the location of the file, and you create (with the <- command) a new dataframe which I am calling `was.counties`, but you can call whatever you'd like.

Make a comment that you are loading .csv data, and bring it in. I write

```
# load csv data
was.counties <- read.csv("h:/pppa_data_viz/2019/tutorial_data/was_msas_1910_2010_20190107.csv")
```

Note that R uses a forward slash to denote directories, even though Windows usually uses a blackslash.

This command creates a dataframe (R's version of a dataset) that contains the input csv file.

D. What's in these data?

The rest of this class teaches you techniques for looking at and summarizing data. You might think, "why can't I just do this in Excel?" Some of what you're learning today may seem easier in Excel... until you get a very large dataset that you can't use easily (or at all) in Excel. We are going to learn techniques that are robust to almost any sample size. We're practicing with a small dataset to begin so you can see all the data and better understand what's going on.

Now that you've loaded these data, let's take a look at them. How big are these data? R's `dim()` command reports the dimension of a dataframe:

```
# how big is it?
print("this is how many rows x columns the dataset has")
```

```
## [1] "this is how many rows x columns the dataset has"
```

```
dim(was.counties)
```

```
## [1] 246   5
```

The output in the console window tells us that this dataframe has `246` rows and `5` columns.

Rows and columns are the building blocks of dataframes. Starting by understanding whether the number of rows and columns you have is reasonable is always a good place to begin. There are about 20 jurisdictions in the Washington area, and the data set has 11 years, so this seems like a reasonable number of rows.

Next, let's explore what variables this dataframe has. We query the variable names with the `names()` command:

```
# what variables does it have?
print("these are the names of the variables")
```

```
## [1] "these are the names of the variables"
```

```
names(was.counties)
```

```
## [1] "statefips"  "countyfips" "cv1"        "year"       "cv28"
```

We learn that this dataframe has five variables. States and counties are identified by FIPS (federal information processing) codes, which you can find at this website (and many others). State FIPS codes are always two digits, and county codes are always three digits.

Each observation in the `was.counties` dataframe has a state FIPS and a county FIPS code. Beware that county FIPS codes are not unique across states. In other words, two states may have a county 003. To uniquely identify a county, you need both the state and county FIPS codes.

The remaining undefined variables here are `cv1`, which is population, and `cv28`, which is the number of housing units.

Apart from the name of the variable, it is also helpful to know whether a variable is numeric (numbers only), or a string (there are many kinds of strings in R, and we'll hold on discussing this till a future class). You can do mathematical operations with numeric variables, but not with strings.

Use `str()` (for structure) to find the types of variables in this dataframe:

```
# what kinds of variables are these?
print("these are the types of variables")
```

```
## [1] "these are the types of variables"
```

```
str(was.counties)
```

```
## 'data.frame':    246 obs. of  5 variables:
##  $ statefips : int  11 24 24 24 24 24 51 51 51 51 ...
##  $ countyfips: int  1 9 17 21 31 33 13 43 47 59 ...
##  $ cv1       : int  331069 10325 16386 52673 32089 36147 10231 7468 13472 20536 ...
##  $ year      : int  1910 1910 1910 1910 1910 1910 1910 1910 1910 1910 ...
##  $ cv28      : int  NA NA NA NA NA NA NA NA NA NA ...
```

This reports that there are five variables. All variables are of type "int" for integer, meaning they are all numeric with no decimals.

To get an overall sense of the magnitude of these variables, use `summary()`, which reports summary statistics on each variable:

```
# look at values
print("these are the values they take on")
```

```
## [1] "these are the values they take on"
```

```
summary(was.counties)
```

```
##    statefips       countyfips         cv1               year
##  Min.   :11.00   Min.   :  1.0   Min.   :   5199   Min.   :1910
##  1st Qu.:24.00   1st Qu.: 31.0   1st Qu.:  13350   1st Qu.:1940
##  Median :51.00   Median : 61.0   Median :  24218   Median :1960
##  Mean   :43.31   Mean   :178.3   Mean   : 119984   Mean   :1962
##  3rd Qu.:51.00   3rd Qu.:179.0   3rd Qu.:  93974   3rd Qu.:1990
##  Max.   :54.00   Max.   :685.0   Max.   :1081726   Max.   :2010
##                                  NA's   :6
```

```
##       cv28
##  Min.   :  1739
##  1st Qu.:  5832
##  Median : 13438
##  Mean   : 58393
##  3rd Qu.: 57275
##  Max.   :407998
##  NA's   :86
```

Here we learn that the `statefips` variable seems to take on only limited values (11,24,51,43), which makes sense: there are four states (counting DC) in the greater Washington metropolitan area. The population variable (`cv1`) has six observations with no value (NA), and the housing variable 86.

While an average is a reasonable way to look at population, it makes less sense for a categorical variable like `statefips` ("categorical" means the variable takes on a number of discrete categories; there is no state 11.5, for example). To look at the distribution of these categorical variables, it may be helpful to make a frequency table, which is easy in R using `table()`:

```
# look at non-numeric variables
print("for non-numeric variables")
```

```
## [1] "for non-numeric variables"
```

```
table(was.counties$statefips)
```

```
##
##  11  24  51  54
##  11  55 169  11
```

We see that there are four unique values for `statefips`. State 11 (DC) has 11 observations (one for each decade). State 24 (Maryland) has 55 observations, or 5 for each year. State 51 (Virginia) has 169 observations; it has a very complicated institutional set-up with many small jurisdictions. State 54 is West Virginia, and has one county that appears 11 times.

E. Dataframe Structure

To work with dataframes, it is exceedingly helpful to know how to refer to rows and columns. The next set of commands gives examples. Generally, you can refer to the rows and columns in a dataframe using `dataframe.name[rows,columns]`. This convention of rows then columns is standard in R.

You can use this format to print rows to the screen. Here I print the first five:

```
# print some rows to the screen
print("first five rows")
```

```
## [1] "first five rows"
```

```
was.counties[1:5,]
```

```
##   statefips countyfips    cv1 year cv28
## 1        11          1 331069 1910   NA
## 2        24          9  10325 1910   NA
## 3        24         17  16386 1910   NA
## 4        24         21  52673 1910   NA
## 5        24         31  32089 1910   NA
```

This shows only the first five rows. You could print rows 20 to 30 by replacing `1:5` with `20:30`.

You can also print just some columns:

```
# print some columns to the screen
print("first two columns")
```

```
## [1] "first two columns"
```

```
was.counties[1:10,1:2]
```

```
##    statefips countyfips
## 1         11          1
## 2         24          9
## 3         24         17
## 4         24         21
## 5         24         31
## 6         24         33
## 7         51         13
## 8         51         43
## 9         51         47
## 10        51         59
```

This prints rows 1 to 10 and columns 1 and 2. To print all rows, with columns 1 and 2, you would write `was.counties[,1:2]` (but this prints 200 rows and takes up too much space for this tutorial!).

Above, I used the column order to pick out columns. Frequently this is not helpful as you may not know the column ordering, or the column ordering many change. You can also use the column name directly, as I do below. Rather than the column name, I tell R to take the columns named `statefips`, `countyfips`, and `year`. I use the notation `c()` to list a vector (list of things) that R should take.

```
# print columns by name
print("first five rows and three columns")
```

```
## [1] "first five rows and three columns"
```

```
was.counties[1:5,c("statefips","countyfips","year")]
```

```
##   statefips countyfips year
## 1        11          1 1910
## 2        24          9 1910
## 3        24         17 1910
## 4        24         21 1910
## 5        24         31 1910
```

F. Sorting

It is frequently helpful to sort when looking at data. The most frequent sort command in R is `order()`. These sorting commands illuminate how you can call particular rows, as we learned above, and how to you refer to a specific variables in a dataframe: `dataframe$var1` for `var1` in `dataframe`.

We'll begin by sorting by the number of housing units. The order command goes inside the row part of the brackets, since we are sorting rows. Note that we have to refer to the variable by the full name inside the square brackets. We replace `was.counties` with the new, sorted version and then print the first five rows.

```
# sort by housing units
print("sort by number of housing units, print first 5 obs")
```

```
## [1] "sort by number of housing units, print first 5 obs"
```

```
was.counties <- was.counties[order(was.counties$cv28),]
was.counties[1:5,]
```

```
##     statefips countyfips  cv1 year cv28
## 94         51        157 6112 1950 1739
## 118        51        157 5368 1960 1865
## 173        51        685 6524 1980 1931
## 142        51        157 5199 1970 2024
## 88         51         43 7074 1950 2097
```

The data are sorted smallest to largest, so we learn that the jurisdiction with the fewest housing units in the DC metro area is Virginia's county 157 (Rappahanock County).

We can also sort from biggest to smallest by putting a - sign in front of the variable. Again, we print the first five, now the largest:

```
# sort by housing units descending
print("sort by number of housing units descending, print first 5 obs")
```

```
## [1] "sort by number of housing units descending, print first 5 obs"
```

```
was.counties <- was.counties[order(-was.counties$cv28),]
was.counties[1:5,]
```

```
##     statefips countyfips     cv1 year   cv28
## 232        51         59 1081726 2010 407998
## 227        24         31  971777 2010 375905
## 208        51         59  969749 2000 359411
## 203        24         31  873341 2000 334632
## 228        24         33  863420 2010 328182
```

We learn that Virginia's county 59 has about 400,000 housing units in 2010, and is the jurisidiction with the most housing units.

It is also possible to sort by two variables. Suppose we'd like to know the number of housing units sorted by year:

```
# sort by year and descending housing units
print("sort by year and descending housing units")
```

```
## [1] "sort by year and descending housing units"
```

```
was.counties <- was.counties[order(was.counties$year,-was.counties$cv28),]
was.counties[1:5,]
```

```
##   statefips countyfips    cv1 year cv28
## 1        11          1 331069 1910   NA
## 2        24          9  10325 1910   NA
## 3        24         17  16386 1910   NA
## 4        24         21  52673 1910   NA
## 5        24         31  32089 1910   NA
```

We learn that in 1910, we do not observe housing units.

G. Subsetting

It is also frequently useful to work with a smaller dataset. (Perhaps not with this current dataset, but the principle we'll learn here will be useful in the future.)

We use the same logic of limiting rows and columns as we did before. To make a dataframe without 1910, we tell R to take all rows where the variable year is not equal (`!=`) to 1910. We check the new dataframe (`was.counties.no1910`) using `dim()`.

Note that we are creating a new dataframe called `was.counties.no1910`. This dataframe exists in addition to the `was.counties` dataframe. The ability to have multiple dataframes is a key strength of R relative to Excel or Stata.

```
# make a dataframe without 1910
print("make a dataset w/o 1910")
```

```
## [1] "make a dataset w/o 1910"
```

```
was.counties.no1910 <- was.counties[was.counties$year != 1910,]
dim(was.counties.no1910)
```

```
## [1] 226   5
```

Recall that the original had 246 rows. Does this seem reasonable?

We can use any variable to subset. Below we omit Washington, DC from the dataframe, again using the not equals command:

```
# make a dataframe without washington dc
print("make a dataframe w/o washington dc")
```

```
## [1] "make a dataframe w/o washington dc"
```

```
was.counties.no.dc <- was.counties[was.counties$statefips != "11",]
dim(was.counties.no.dc)
```

```
## [1] 235   5
```

Subsetting is not just limited to rows. To remove a column, I can tell R to not include any column where the column name is cv28: `!(names(was.counties) %in% ("cv28"))`.

```
# make a dataframe without housing (cv28)
print("make a dataframe w/o housing variable")
```

```
## [1] "make a dataframe w/o housing variable"
```

```
was.counties.no.cv28 <- was.counties[, !(names(was.counties) %in% ("cv28"))]
dim(was.counties.no.cv28)
```

```
## [1] 246   4
```

Note that the number of rows is still 246, but the number of columns is now 4, rather than 5.

H. Create new variables and dataframes

In this step, we create a new variable and dataframes of summary statistics.

1. Create a new variable

Creating a new variable in R is reasonably straightforward. Use the `<-` to denote the output, and use the `dataframe$var` notation to describe variables.

For example, to divide one variable by another, do the below.

```
# people per housing unit
print("make new variable with people per housing unit in each county")
```

```
## [1] "make new variable with people per housing unit in each county"
```

```
was.counties$ppl.per.hu <- was.counties$cv1 / was.counties$cv28
summary(was.counties$ppl.per.hu)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   1.909   2.566   2.762   2.852   3.272   3.975      86
```

Here I create the number of people per housing units, and then I use `summary()` to check the value. When programming, it is always a good idea to build in sanity checks as things can and do go wrong.

2. Summarizing data

The most familiar summary statistic is the mean. R can easily calculate the mean of a variable with `mean()`. If there are any missing values in a variable and you don't tell R to omit these missing value observations, R will report a missing value for the mean. That is why the first mean command below reports `NA`, while the second gives a value. The `na.rm = TRUE` means "yes, omit missing values in calculations."

```
# brute force
print("find a mean")
```

```
## [1] "find a mean"
```

```
mean(was.counties$cv1)
```

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

```
mean(was.counties$cv1, na.rm = TRUE)
```

```
## [1] 119983.7
```

Does this value seem reasonable for the mean jurisdiction population over the entire period?

It is also sometimes helpful to put the mean into its own variable (maybe you'll want to put it in a title or some such). You can do this as we do below, creating a `newvar`.

```
# putting variable into its own variable
print("put mean value into a variable")
```

```
## [1] "put mean value into a variable"
```

```
newvar <- mean(was.counties$cv1, na.rm = TRUE)
newvar
```

```
## [1] 119983.7
```

There are **many** different statistical functions in R – so many that you're better off googling "statistical functions in R" to think about this. Most of them work just like the `mean()` function that we just explored.

We now move on to creating dataframes of summary values. Suppose we'd like to know the mean population by year. We can do this using a command called `summarize`, which is part of the tidyverse, a set of R packages that we'll come to know better thoughout the course of the semester.

What you need to know now is that you need a new package (a package is a set of commands written by someone to plug into R). The first time you use a new package you need to install it, as in

```
install.packages("dplyr")
```

You should do this now, but will not need to do it again. Then, having installed the package, you need to let R know that you want to use it:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

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

You should put this command at the top of your R script, so R will load the package when you start running the program. We will use two commands from the `dplyr` package (for now – it contains many other useful commands): `group_by` and `summarize`.

The `group_by` command tells R by which variables you'd like to group your data. Once you've told R, you can then turn to `summarize` to make summary statistics by the groups you've defined. The command `group_by` doesn't change the look of your data – it just changes its functionality in susequent commands.

```
# summarize by year
print("find average by year")
```

```
## [1] "find average by year"
```

```
was.counties.grp.yr <- group_by(was.counties,year)
was.by.year <- summarize(was.counties.grp.yr,cv1.yr=mean(cv1))
was.by.year
```

```
## # A tibble: 11 x 2
##     year  cv1.yr
##    <int>   <dbl>
##  1  1910  32892.
##  2  1920  39282.
##  3  1930  44222.
##  4  1940  59891.
##  5  1950     NA
##  6  1960     NA
##  7  1970 143834.
##  8  1980 142777.
##  9  1990 173222.
## 10  2000 201560.
## 11  2010 234843.
```

Looking at this new dataframe, we have one observation per year (correct!), with a population mean that is increasing over time (seems reasonable). Unfortunately, we have some missing values. We can correct by using the same `na.rm=TRUE` as above:

```
# summarize by year w/o missings
print("find average by year w/o missing values")
```

```
## [1] "find average by year w/o missing values"
```

```
was.by.year <- summarize(was.counties.grp.yr,cv1.yr=mean(cv1, na.rm = TRUE))
was.by.year
```

```
## # A tibble: 11 x 2
##     year  cv1.yr
##    <int>   <dbl>
##  1  1910  32892.
##  2  1920  39282.
##  3  1930  44222.
##  4  1940  59891.
##  5  1950  81966.
##  6  1960 110812.
##  7  1970 143834.
##  8  1980 142777.
##  9  1990 173222.
## 10  2000 201560.
## 11  2010 234843.
```

An improvement!

This set-up is flexible. We can calculate not just the mean population, but also the total population by adding an additional function (`sum()`).

```
# summarize two variables by year
print("find two things by year")
```

```
## [1] "find two things by year"
```

```
was.by.year <- summarize(was.counties.grp.yr,cv1.yr=mean(cv1, na.rm = TRUE),
                                             cv.yr.total = sum(cv1, na.rm = TRUE))
was.by.year
```

```
## # A tibble: 11 x 3
##     year  cv1.yr cv.yr.total
##    <int>   <dbl>       <int>
##  1  1910  32892.      657845
##  2  1920  39282.      785643
##  3  1930  44222.      884441
##  4  1940  59891.     1197826
##  5  1950  81966.     1721291
##  6  1960 110812.     2327056
##  7  1970 143834.     3164346
##  8  1980 142777.     3426648
##  9  1990 173222.     4157327
## 10  2000 201560.     4837428
## 11  2010 234843.     5636232
```

Additionally, we can add a second variable to `group_by`. Instead of summaries by year, we can report data by state and year. Notice how we first define a new "grouped" dataframe.

```
# summarize by state and year
print("find info by state and year")
```

```
## [1] "find info by state and year"
```

```
was.counties.grp.st <- group_by(was.counties,year,statefips)
was.by.state.yr <- summarize(was.counties.grp.st,
                      cv.st.total = sum(cv1, na.rm = TRUE))
was.by.state.yr
```

```
## # A tibble: 44 x 3
```

```
## # Groups:   year [?]
##     year statefips cv.st.total
##    <int>     <int>       <int>
##  1  1910        11      331069
##  2  1910        24      147620
##  3  1910        51      163267
##  4  1910        54       15889
##  5  1920        11      437571
##  6  1920        24      158258
##  7  1920        51      174085
##  8  1920        54       15729
##  9  1930        11      486869
## 10  1930        24      189435
## # ... with 34 more rows
```

Now, rather than 11 observations, we have 44. Does that make sense?

I. Problem Set 1

Now you are ready to work on your own.

For this and all subsequent problem sets you need to turn in one pdf that includes

- R script for the tutorial and these questions (the .R program file)
- R output (from console window; ok to paste into a separate file)
- written output that directly answers the questions (not just the code that finds the numbers)

Turn in to the google folder (class google folder, homeworks, problem set 1) before next class. Save as [last name]_PS1.pdf

You are welcome and encouraged to work with others on the homework. However, each of you must turn in your own homework, in your own words. All duplicate versions of a homework receive a grade of zero.

1. Find the average population in DC for the entire period 1910-2010
2. Find state-level (or the part of the state we observe) average population over the entire period. Write the results in a sentence or two.
3. For each of the four states, are there more or fewer jurisdictions in this dataset now than in 1910?
4. What is the most populous jurisdiction in the DC area in 2010?