Data Visualization Notes

Tim Riffe

2019-01-02

Data Visualization Notes

This is a starter RMarkdown template to accompany *Data Visualization* (Princeton University Press, 2019), which will be the reference book used for the first half of the course *Demographic Exploration and Discovery (with R)*. You can use it to take notes, write your code, and produce a good-looking, reproducible document that records the work you have done. This is a document that we will add to incrementally, live typing in class. My version won't look like your version, but you'll be able to sync with my version if you so desire. At the very top of the file is a section of *metadata*, or information about what the file is and what it does. The metadata is delimited by three dashes at the start and another three at the end. You should change the title, author, and date to the values that suit you. Keep the `output` line as it is for now, however. Each line in the metadata has a structure. First the *key* ("title", "author", etc), then a colon, and then the *value* associated with the key.

This is an RMarkdown File

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

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. A *code chunk* is a specially delimited section of the file. You can add one by moving the cursor to a blank line choosing Code > Insert Chunk from the RStudio menu. When you do, an empty chunk will appear:

```r
# here is a piece of code.
#  Now this code chunk appears
#  in the final document
a <- c(1,2,3)
a
```

```
## [1] 1 2 3
```

```r
# this is a different chunk of code
# in this case just comments
```

Code chunks are delimited by three backticks (found to the left of the 1 key on US and UK keyboards) at the start and end. The opening backticks also have a pair of braces and the letter `r`, to indicate what language the chunk is written in. You write your code inside the code chunks. Write your notes and other material around them, as here.

Before you Begin

To install the tidyverse, make sure you have an Internet connection. Then *manually* run the code in the chunk below. If you knit the document if will be skipped. We do this because you only need to install these packages once, not every time you run this file. Either knit the chunk using the little green "play" arrow to the right of the chunk area, or copy and paste the text into the console window.

```
## This code will not be evaluated automatically.
## (Notice the eval = FALSE declaration in the options section of the
## code chunk)

my_packages <- c("tidyverse", "broom", "coefplot", "cowplot",
                 "gapminder", "GGally", "ggrepel", "ggridges", "gridExtra",
                 "here", "interplot", "margins", "maps", "mapproj",
                 "mapdata", "MASS", "quantreg", "rlang", "scales",
                 "survey", "srvyr", "viridis", "viridisLite", "devtools")

install.packages(my_packages, repos = "http://cran.rstudio.com")
```

Set Up Your Project and Load Libraries

In the first session we managed to get most things installed. If you need to catch up on that, look at `Setup.pdf` from the lecture folder `Lecture_03042019`. We will start using some of those packages already in this lesson. To begin we must load some libraries we will be using. If we do not load them, R will not be able to find the functions contained in these libraries. The tidyverse includes ggplot and other tools. We also load the `socviz` and `gapminder` libraries.

Notice that here, the braces at the start of the code chunk have some additional options set in them. There is the language, `r`, as before. This is required. Then there is the word `setup`, which is a label for your code chunk. Labels are useful to briefly say what the chunk does. Label names must be unique (no two chunks in the same document can have the same label) and cannot contain spaces. Then, after the comma, an option is set: `include=FALSE`. This tells R to run this code but not to include the output in the final document.

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

```
# this just renders the object gapminder in the console
gapminder
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Afghanistan Asia       1957    30.3  9240934      821.
##  3 Afghanistan Asia       1962    32.0 10267083      853.
##  4 Afghanistan Asia       1967    34.0 11537966      836.
##  5 Afghanistan Asia       1972    36.1 13079460      740.
##  6 Afghanistan Asia       1977    38.4 14880372      786.
##  7 Afghanistan Asia       1982    39.9 12881816      978.
##  8 Afghanistan Asia       1987    40.8 13867957      852.
##  9 Afghanistan Asia       1992    41.7 16317921      649.
## 10 Afghanistan Asia       1997    41.8 22227415      635.
## # ... with 1,694 more rows
```

The remainder of this document contains the chapter headings for the book, and an empty code chunk in each section to get you started. Try knitting this document now by clicking the "Knit" button in the RStudio toolbar, or choosing File > Knit Document from the RStudio menu.

everything has names

Here some first words on names, objects, and functions.

```r
my_numbers <- c(1,2,3,4,5)
my_numbers
```

```
## [1] 1 2 3 4 5
```

```r
# mean is a named function (all functions have names)
# I can get the mean of my numbers by feeding them in
mean(my_numbers)
```

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

```r
# extend the vector by adding in an NA
my_numbers2 <- c(my_numbers, NA)
# mean() gives different answer, why?
mean(my_numbers2)
```

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

```r
# use the help!
?mean
# ahhhh there's an argument to throw out NAs
mean(my_numbers2, na.rm = TRUE)
```

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

```r
# this line provokes a simple error:
#mean()
```

Now I'm going to copy and paste some example code from `?mean` to see how to get a feel for working with a *strange and exotic* function that we've never seen before and have no idea how to use.

```r
x  <- c(0:10, 50)
xm <- mean(x)
xm
```

```
## [1] 8.75
```

```r
c(xm, mean(x, trim = 0.10))
```

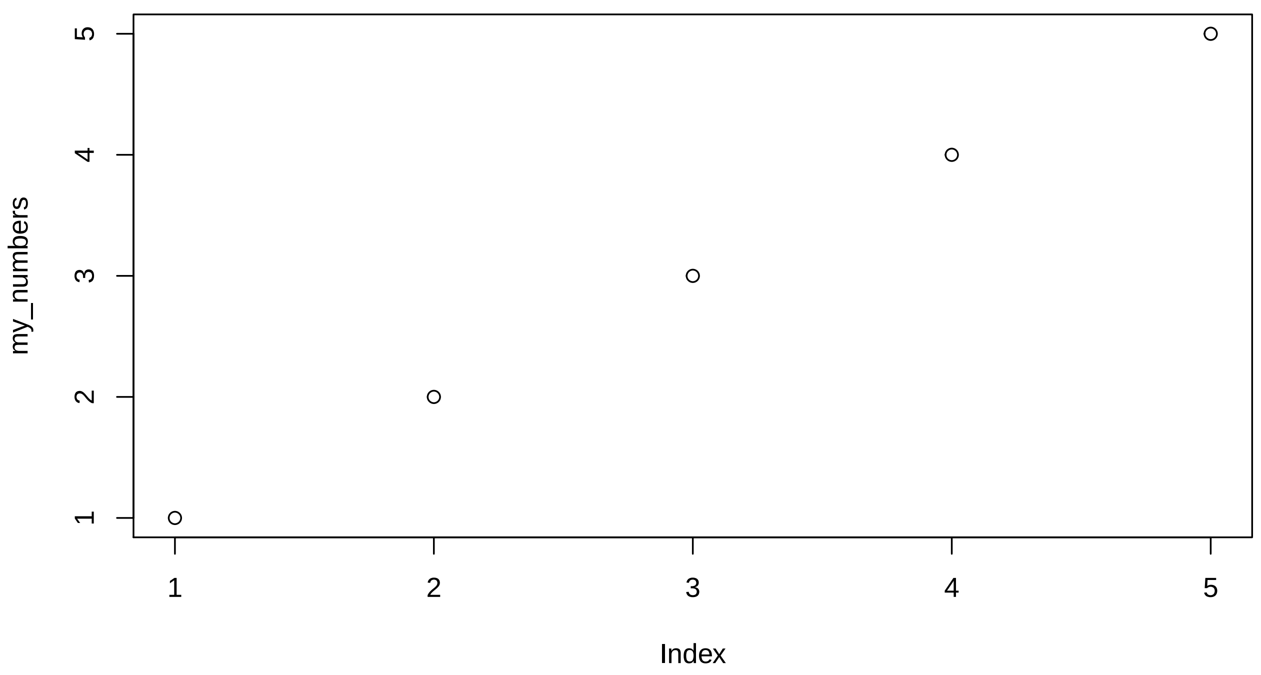
```
## [1] 8.75 5.50
```

Maybe between the act of running this code and actually reading the help file we can figure out what the `trim` argument does.

Make a Plot

If you generate a plot, it gets included in the final output automatically, which is great!

```r
plot(my_numbers)
```



## what kind of object do we have?

`str()` tells us what an object is in and gives a taste of what's in it. `view()` opens an object to the extent possible in a viewer. Only makes sense for tabular data IMO. DANGER don't edit data by hand in `R`, and avoid it elsewhere too.

```r
str(titanic)
```

```
## 'data.frame':    4 obs. of  4 variables:
##  $ fate   : Factor w/ 2 levels "perished","survived": 1 1 2 2
##  $ sex    : Factor w/ 2 levels "female","male": 2 1 2 1
##  $ n      : num  1364 126 367 344
##  $ percent: num  62 5.7 16.7 15.6
```

```r
view(titanic)
```

At this point Tim tried to wax philosophical about the virtues of tidy programming, visible programming (that is, formulas not hidden from view), forgetful `R` sessions, and things of this nature. Now a couple other primitive lessons.

`R` coerces objects to the lowest common denominator.

```r
# integer vs double. Double wins!
ints <- as.integer(c(1,2,3))
str(ints)
```

```
##  int [1:3] 1 2 3
```

```r
str(c(ints, pi))
```

```
##  num [1:4] 1 2 3 3.14
```

```r
# character beats integer / numeric etc
str(c(ints, "a character string"))
```

```
##  chr [1:4] "1" "2" "3" "a character string"
```

What can we do with vectors. Everything (?) is a vector? I think everything is a vector.

```
# elementwise operations
ints * 2 # with a scalar
```

```
## [1] 2 4 6
```

```
ints + 2
```

```
## [1] 3 4 5
```

```
ints ^ 2
```

```
## [1] 1 4 9
```

```
# elementwise with an object of equal dimension
ints * ints
```

```
## [1] 1 4 9
```

Let's generalize this statement. Matrices are vectors?

```
# class ended here! Oops.
```