Lab 2: Data and Efficiency

Matthew Varkony

1/19/2020

As I mentioned in our first lab, learning to program takes patience and exposure. In this lab we will introduce some of the basics of `R` programming including data structures, function creation and automation. The goal of using statistical programming is to investigate quantitative questions in a robust and efficient manner. It is important to have a general understanding of the how `R` reads and interprets the data for analysis before we begin creating or using functions to analyze our data. The accessibility of data is growing at break neck speeds, and in order to make sense of all this information utilizing the speed of computers with automated tasks is necessary.

Data and its Structure

Loading Packages and Data

```
library(tidyverse)
```

```
##install.packages("tidyverse")
##library("tidyverse")
##install.pacakges(tidylog)
library(tidylog)
```

```
##
## Attaching package: 'tidylog'
```

```
## The following objects are masked from 'package:dplyr':
##
##     add_count, add_tally, anti_join, count, distinct, distinct_all,
##     distinct_at, distinct_if, filter, filter_all, filter_at, filter_if,
##     full_join, group_by, group_by_all, group_by_at, group_by_if,
##     inner_join, left_join, mutate, mutate_all, mutate_at, mutate_if,
##     rename, rename_all, rename_at, rename_if, right_join, sample_frac,
##     sample_n, select, select_all, select_at, select_if, semi_join,
##     slice, summarise, summarise_all, summarise_at, summarise_if,
##     summarize, summarize_all, summarize_at, summarize_if, tally,
##     top_frac, top_n, transmute, transmute_all, transmute_at,
##     transmute_if, ungroup
```

```
## The following objects are masked from 'package:tidyr':
##
##     drop_na, fill, gather, pivot_longer, pivot_wider, replace_na,
##     spread, uncount
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

```
##install.pacakages(data.table)
data <- data.table::fread("lab2_data.csv",
                          colClasses = c(pin = "character"))
```

```
## Warning in data.table::fread("lab2_data.csv", colClasses = c(pin =
## "character")): Column name 'pin' (colClasses[[1]][1]) not found
```

As we can see with the output form tidylog many of the functions of `dplyr` and `tidyr` (from the `tidyverse`) are "masked" after loading `tidylog` into our library space. When something is masked it means that the functions in one package have the same name as functions in another package. The functions from the package that is loaded in most recently are operated with if their is a masking conflict. That is why it is sometimes smart to not even call in packages with `library`, and instead we can call on a function directly from a package. An example of this is when we read in the data, we use `data.table::fread` to avoid loading the package `library(data.table)` into our work space environment. It is good practice to do this with functions that you do not use often in your code.

We can check out the functions within a package directly through `R` or by going on the internet and looking for the CRAN (Comprehensive R Archive Network) documentation. I think the easiest way is to just search the package on the internet, most packages that we use starting out are in the CRAN library and have a specific web page dedicated to helping us use the package.

```
# help(dplyr)

# ?dplyr

# vignette("dplyr")
```

Typing in `plm r cran` to Google leads us to the `plm` CRAN page. Here we can see all the information we need about the package. The reference manual is the best place to get the technical information on functions and their arguments within a package.

We'll start by showing the different ways to create vectors and other data structures. Than we will go back to our loaded in data frame to see the different types of data structures that are used in `R`.

```
vec1 <- c(1,2,3,4,5,6,7,8,9,10) ##important to recognize that the c(...) is used to start a vector
vec2 <- 1:10 ##don't need the c(...) because R creates a vector in bgd
veca <- c("a", "b", "c", "d") ##since this is a vector of characters we need the c(...)
vec3 <- seq(1,10, by = 1) ##again some commands inhernetly adopt the c(...) vector structure
vec1
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
vec2
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
vec3
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

It's important to remember that there is not just one correct way to achieve an outcome in `R`. The method that you chose depends on your familiarity with the commands, your desired outcomes, and the flow of your code. As you get better at programming focusing on code efficiency will make a big difference in the commands that you run. For learning about code efficiency it is a good idea to check the out the book by Colin Gillespie and Robin Lovelace.

We can also combine vectors, together to create new vectors, as well as to create data frames. The data frame is the most important object that you will work with in `R`. It can be thought of as a "row" or "column" combinations of vectors. The data set that we previously loaded in is a data.frame, where each column represents a vector.

Below is an example of matrix, an array, and a data frame.

```r
mat1 <- matrix(1:10, nrow = 5, ncol = 2)
mat2 <- matrix(c(vec1[1:5], vec1[6:10]), 5, 2)
arr1 <- array(c(mat1, mat2), dim = c(5,2,2))
df <- data.frame(1:5, 6:10)
mat1
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

```r
mat2
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

```r
arr1
```

```
## , , 1
## 
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
## 
## , , 2
## 
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

```r
df
```

```
##   X1.5 X6.10
## 1    1     6
## 2    2     7
## 3    3     8
## 4    4     9
## 5    5    10
```

We can also create matrices by binding together row or column vectors.

```r
vec4 <- c(10:15)
vec5 <- rep(5, 6)
rbind(vec4, vec5)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## vec4   10   11   12   13   14   15
## vec5    5    5    5    5    5    5
```

```
cbind(vec4, vec5)
```

```
##      vec4 vec5
## [1,]   10    5
## [2,]   11    5
## [3,]   12    5
## [4,]   13    5
## [5,]   14    5
## [6,]   15    5
```

You probably won't be using arrays in this class, but it is important to know that you can create a "list" of matrices or vectors.

The above examples are a subset of vectors called "atomic vectors", while the other type of vectors are called "lists". These are both vectors, however the structure of these "vectors" is different. Atomic vectors all must have the same type of elements. While in a list vector the elements can be of different classes.

The 4 Types of Atomic Vectors

1. Logical

```
a <- T
b <- FALSE
a
```

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

```
b
```

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

```
is.logical(a)
```

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

```
typeof(a)
```

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

2. Numeric

a. Doubles: Can be decimals 2.345 or scientific 1.25×10^5. + 3 Special Cases
   - `Inf`
   - `-Inf`
   - `NaN` (not a number)

```
c <- 2.45
typeof(c)
```

```
## [1] "double"
```

```
is.double(c)
```

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

```
is.integer(c)
```

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

```r
is.numeric(c)
```

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

b. Integers: Written similar to doubles, but followed by an L, 2

```r
d <- 1234L
typeof(d)
```

```
## [1] "integer"
```

```r
is.double(d)
```

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

```r
is.integer(d)
```

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

```r
is.numeric(d)
```

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

Most of the time you will be using doubles. Almost exclusively I would say.

4. Characters or Character Strings: This is how we right words in `R`

```r
e <- "e" ##this is a character
typeof(e)
```

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

```r
is.character(e)
```

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

```r
f <- "Hi my name is Matt" ##this is a character string
typeof(f)
```

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

- Important that when we check for NA's we use `is.na()` command. Since NA's will give you headaches due to the fact that they are contagious. Always be careful, and when in doubt, if something doesn't make sense check for NAs

```r
g <- c(NA, 1, NA, 2)
g == NA ##this won't work because no reason to believe one NA = to another NA
```

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

```r
is.na(g)
```

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

We don't have the time currently to get into the details for the attributes of vectors. But if you are interested in reading more about assigning attributes to vectors check out this chapter.

I will mention one attribute that is important for work that we will be doing. The class of an atomic vector determines how `R` uses the vector we are working with. One specific class that is important is a **factor**. This is for use with categorical variables such as "male" or "female" sex designations. We can coerce these designations to factors from their original class as character strings.

```r
sex <- c("male", "female", "female", "female")
typeof(sex)
```

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

```
class(sex)
```

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

```
##now if we coerce this variable into a factor
sex.f <- as.factor(sex)
typeof(sex.f)
```

```
## [1] "integer"
```

```
class(sex.f)
```

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

```
sex.f
```

```
## [1] male   female female female
## Levels: female male
```

We see that the factor now works as a type of integer (this is for computational efficiency). Since there are only two different types of character strings there are only two levels. This is just a quick introduction to factors and should serve the purpose of this class. But if you are interested in learning more about vectors check out this page.

List Vectors

Lists can be thought of as vectors with an assortment of different classes in them.

```
l1 <- list(logical = c(TRUE, FALSE, FALSE),
           double = c(1:4),
           integer = c(4L),
           character = c("Matt V")
           )
l1
```

```
## $logical
## [1]  TRUE FALSE FALSE
##
## $double
## [1] 1 2 3 4
##
## $integer
## [1] 4
##
## $character
## [1] "Matt V"
```

As a quick note it is also nice to think of data frames as just a list of vectors, as was mentioned earlier. We

Now let's get back to the data set that we loaded in at the beginning of class. We will explore this data set and create some functions to use with this data set.

Before we start any analysis with data set it is a good idea to get familiar with its contents.

```
##this checks the dimensions of our data
dim(data) ##1000 rows x 62 columns
```

```
## [1] 1000   20
```

```r
##to get the names of our columns we can use
names(data)
```

```
##  [1] "sale.amount1"      "sale.amount2"      "sale.amount3"
##  [4] "sale.year1"        "sale.year2"        "sale.year3"
##  [7] "sale.date1"        "sale.date2"        "sale.date3"
## [10] "beds"              "baths"             "stories"
## [13] "yr.built.actual"   "total.sqf"         "land.use.fdor.dec"
## [16] "elev"              "coast.d"           "fzone"
## [19] "lon"               "lat"
```

```r
##now to check the structure of our variables we can use
str(data)
```

```
## Classes 'data.table' and 'data.frame':   1000 obs. of  20 variables:
##  $ sale.amount1     : num  70000 120000 100 0 134000 172000 450000 300000 0 322000 ...
##  $ sale.amount2     : num  59000 71500 200000 39000 0 ...
##  $ sale.amount3     : num  33400 0 0 33800 76000 ...
##  $ sale.year1       : int  1986 2004 2018 1980 2003 2005 2017 1999 1998 2015 ...
##  $ sale.year2       : int  1981 2000 2018 1977 1996 1997 2005 1996 1985 2013 ...
##  $ sale.year3       : int  1975 1993 2006 1974 1995 1991 NA NA 1979 2010 ...
##  $ sale.date1       : chr  "1986-10-01" "2004-01-01" "2018-07-31" "1980-07-01" ...
##  $ sale.date2       : chr  "1981-02-01" "2000-11-01" "2018-06-29" "1977-10-01" ...
##  $ sale.date3       : chr  "1975-04-01" "1993-10-01" "2006-04-01" "1974-01-01" ...
##  $ beds             : int  3 2 2 3 3 0 2 4 4 2 ...
##  $ baths            : int  2 2 2 2 2 0 2 3 2 2 ...
##  $ stories          : int  1 0 0 2 0 0 0 1 1 0 ...
##  $ yr.built.actual  : int  1954 1982 1971 1973 1977 1987 2004 1950 1979 2010 ...
##  $ total.sqf        : int  1053 991 1266 1378 1275 825 1332 4523 2444 888 ...
##  $ land.use.fdor.dec: chr  "Single Family Residential" "Condominiums" "Condominiums" "Condominiums"
##  $ elev             : num  2.29 1.66 3.01 1.92 1.9 1.48 1.37 1.67 2.21 3.63 ...
##  $ coast.d          : num  1956 992 11222 6888 3094 ...
##  $ fzone            : chr  "X" "AE" "AH" "AH" ...
##  $ lon              : num  -80.3 -80.3 -80.4 -80.4 -80.4 ...
##  $ lat              : num  25.9 25.9 25.7 25.7 25.8 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

Loops

`For loops` are used to iterate a process a chosen number of times. As I explained earlier a goal of programming is to automate tasks. The `for loop` is a good way to do this. However, be careful when using `for loops` because they are computationally intensive for some tasks. Sometimes it is better to implement one of the `apply` family functions. However, we will save the `apply` family for next lab.

A simple `for loop` looks like this.

```r
for(i in 1:3){
  print(i)
}
```

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

Important to understand how for loops work, because they may not return what you are expecting

```r
for(j in 1:3){
  j
}
j
```

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

Wee see here that j is assigned the value of the last iteration of the loop. In order to store all the objects that are created with the `for loop` we need to create an empty vector outside of the loop for the variables to be stored in.

```r
means <- c(1, 5, 10)
# ?vector
out <- vector("list", length(means))
for(l in 1:length(means)){
  out[[l]] = rnorm(10, means[l])
}
##run this command below to check what the output looks like
# out
```

One final thing important to know about `for loops` is the ability to alter the output of the loop as it is working through the given set. Here we introduce `breaks` and `ifelse` statements. We can tell `R` to do something else if the loop encounters a certain value.

```r
for (i in 0:5)
  {

  x = i + 5

  # but ...
  if(i == 3) {x = "NULL"}  # our new if statement

  cat(paste("i= ", i, "and i + 5 =", x,"\n"))

  } # end for loop
```

```
## i=  0 and i + 5 = 5
## i=  1 and i + 5 = 6
## i=  2 and i + 5 = 7
## i=  3 and i + 5 = NULL
## i=  4 and i + 5 = 9
## i=  5 and i + 5 = 10
```

```r
# Alternatively, This loop makes a bit more sense as a flow of code.
for (i in 0:5)
  {

  if(i == 3) {x = "NULL"} else {x = i + 5} # our new if else statement

  cat(paste("i= ", i, "and i + 5 =", x,"\n"))

  } # end for loop
```

```
## i=  0 and i + 5 = 5
## i=  1 and i + 5 = 6
## i=  2 and i + 5 = 7
## i=  3 and i + 5 = NULL
## i=  4 and i + 5 = 9
## i=  5 and i + 5 = 10
```

Here is an example of a `break` statement.

```
for (i in 0:5)
  {

  x = i + 5 # Do something to i, iteratively

  if(x > 7) {break} # Here is that "break" command

  cat(paste("i= ", i, "and i + 5 =", x,"\n"))

  } # end loop
```

```
## i=  0 and i + 5 = 5
## i=  1 and i + 5 = 6
## i=  2 and i + 5 = 7
```

This is a good introduction to `for loops`. Typically they are used for generating data or running bootstrap analysis.

Functions

Writing functions is another good way to automate your code. They say that anything you need to do more than 3 times should have a function written for it. While I'm not sure there is a hard and fast number attached to this logic, I think it is important to plan your analysis. If you see that there is an opportunity to write a function this is a good idea and will save you time in the long run. Because at the end of the day you can copy that functions code to other assignments in the future, so the dividends keep paying off.

Functions contain 3 main components 1. `formals()` which are the arguments that you use in the call to the function 2. `body()` which is the code the function contains 3. `environment()` contains the data where the function searches for the values associated with the names in the function

As an example we can create a function that converts Celsius to Fahrenheit

```
cel.to.far = function(celsius){
 far = (9/5)*celsius + 32
 return(far)
}
cel.to.far(7)
```

```
## [1] 44.6
```

Examples

Let's start by making a function to determine what our final grade is in this class.

Things to think about:

1. What are the arguments/inputs?
2. What is the task we want to accomplish or calculation to be made using these arguments?
3. What do we expect our function to return?

Now using this function we just created, let's calculate the final grade for the students the 8 students in a class.

```
## Student 1 has a grade of  83
## Student 2 has a grade of  73
## Student 3 has a grade of  69.5
## Student 4 has a grade of  58
## Student 5 has a grade of  68
## Student 6 has a grade of  82.5
## Student 7 has a grade of  55
## Student 8 has a grade of  72
```

Resources for Practice

I suggest if you are not currently comfortable using `R` that you practice a bit on your own using the `swirl` package. This link is for the **R Programming** course. If you are crunched for time there is a fast introduction to `R` from another course in `swirl`, which you can located here. The cool thing about `swirl` is that it is all interactive within the `R` environment. So you run and practice the course within `R`. There are a number of other courses at swirl that you may be interested in if you already feel comfortable with `R`.

It's difficult for me to cover all things that you may need with your homework (and outside work) within the hour, so I would highly recommend looking into some of these resources yourself. This will make it much easier when you are working through the problem sets in this class and other classes requiring `R` experience.

As a note much of the material that is covered in this lecture comes directly from Hadley Wickham's Advanced R book which you can find online at the link provided. There is much more information in this book than we cover, but it is a good resource for further dives into `R`. While this book is a bit more advanced, you can check out this website for a more basic primer of `R` information.