An Introduction to R[1]

Sam Fuller sjfuller@ucdavis.edu

1.1 Getting R

R is available for download from CRAN (Comprehensive R Archive Network): Cran.r-project.org. You'll want to install the most recent (compatible) version. If installing on Windows, I recommend also downloading and installing Rtools (this is optional, but comes in handy if you ever need to compile or test a package).

I highly recommend the use of a text (or line) editor, namely RStudio (Rstudio.com). However, feel free to use base-R or another editing program like Atom (Atom.io) if that's more your speed.

1.2 The R Language

R is an object-oriented language: you can store and manipulate objects with the command <-.[2] For instance:

```
A <- 1          #Assign A the value of 1
A               #Display the value of A
A + 2           #Display, but not assign, the value of A plus 2.
A <- A + 1      #Assign A the value of itself plus one (i.e. add one to A).
B <- A/3        #Assign B the value of A divided by 3.
C <- A * B      #Assign C the value of A times B.
D <- C/B        #Assign D the value of C divided by B.
E <- D          #Assign E the value of D (i.e. make a copy of D called E)
A <- 10         #Reassign/overwrite the value of A with 10.
```

As you can see, you can manipulate objects in a number of ways, namely any basic mathematical operation you can think of. Objects can be manipulated by other objects, by simple numbers, and can be updated as in the `A <- A + 1` example. Also, it's good programming practice to annotate code with comments. This can be done with the use of hashtags or pound signs, which tell R not to execute that particular line and is what I've been using to explain commands like those above. Here is an example where even though a command is written, the code does not actually run because of the hashtag (#):

```
# this is a comment
# 1 + 1
```

[1]Special thanks to Chris Hare for the document from which this lab was built.
[2]Sometimes you'll see a single = instead of <-, this is old syntax and is relatively deprecated since most functions use = to denote arguments. Consequently we recommend against using this syntax, but you may see it when Googling for help or in older texts.

Objects also have different types (classes). For example, the objects above are `numeric`. The most common classes of objects are:

```
#Numeric
A <- 1
#Character/String
text <- "I sure do love coding in R"
#Factor
experiment.dat <- as.factor(c("treated","control"))
#Logical
X <- TRUE; Y <- FALSE #Note: semicolons (;) act the same as linebreaks
#NA (We'll cover these later, they're special)
Z <- NA
#Vector (These commands/objects are explained below)
V1 <- c(1,2)
V2 <- c(0,6)
#Data Frame (We'll get into the specifics of these later)
df <- cbind.data.frame(V1,V2); df <- data.frame(V1,V2)
```

1.2.1 Logical Statements

R can also evaluate logical statements with the syntax `==` (exactly equal to), `!=` (not equal to), `>` (greater than), `>=` (greater than or equal to), `%in%` (in/an element of), etc.

```
3==2
3 > 2
G <- 3
G > 2
G == 2
H <- c(3,6,9,12)
3 %in% H    #This is used with vectors mostly, which we will cover soon.
10 %in% H

Names <- as.factor(c("Charlie", "Dennis", "Mac")) #We can also use these with factors.
"Charlie" %in% Names
"Dee" %in% Names
c("Charlie","Dennis", "Mac", "Dee", "Frank") %in% Names

G >= 2 & G < 4  #Evaluates if G satisfies both arguments.
G == 3 | G == 4 #Evaluates if G satisfies either argument.
G %in% c(3,4)   #This is equivalent to the line above.
```

These statements are incredibly useful for subsetting data based on different values and can be integral for the coding of functions that manipulate data.

[2] `;` is used to separate commands that are on the same line.

1.2.2 Missing Values

Missing values have a special designation (`NA`) in `R`:

```
G <- c(NA,2,3,4)
is.na(G)         #Evaluates whether each element of G is an NA.
!is.na(G)        #Evaluates whether each element of G is not an NA.
na.omit(G)       #Omits all values in G that are NA.
mean(G)          #Mean of G, but returns an NA.
mean(G, na.rm=T) #Mean of G, removes NA values in both the numerator and denominator.
```

`NA`s can cause significant problems when coding certain functions.[3] This is why many functions have an argument `na.rm = T` or `use = "complete.obs"` that removes any NAs in the calculation.

1.2.3 Vectors, Matrices, & Dataframes (Lists Too)

Data isn't just simply stored in single-value, separate objects, however. In the real world, we would want to aggregate multiple values into a single object. For example, if we have the sociodemographic characteristics of a set of individuals (e.g. income, race, gender) we would want to store each characteristic (variable) in its own aggregated object. This leads us to vectors. Simply put, vectors aggregate data into a single object:

```
B <- c(1,2)      #This "concatenates" the values 1 and 2 into a single object B.
B                #Displays B.
A <- c(2,-1)     #You can concatenate any value, including negatives.
D <- A + B       #This adds the ith value of A to the ith value of B resulting in the ith value of D.
E <- c(A,B)      #This combines A and B, sequentially with B following A, into a single object E.

#This combines two strings into a single object.
lyrics <- c("Slip Like Freudian",
            "Your first and last step to playing yourself like accordion")
Answer.Key1 <- c(T,F,T,F)   #You can concatenate logical objects too.
Answer.Key2 <- as.factor(c("A","A","C","D"))   #Same with factors.
text.numeric <- c("two",2)  #A vector's components must all be of the same type (class).
                            #This will coerce (change) the number 2 into a character "2".
```

`c` is short for concatenate and is one of the primary ways to create a vector in `R` (you will use this A LOT). Using `c` essentially aggregates different, distinct values into a single object (vector). Other functions such as `sequence()` or `rep()` also create vectors, which we may cover in the future.

But what if we want to aggregate data in an even larger, more organized format? What if we want to have both the aggregated information of the different values of a vector (e.g. all the different incomes in our sample) *and* the combinations of values from all of the vectors each observation has (e.g. the combination of income, race, and gender for each individual)? This is where matrices/dataframes come in.[4]

[3]For example, NAs can break `if() else()` statements if `is.na()` is not evaluated first.

[4]We will focus on dataframes, rather than matrices, as they are much more commonly used in data science. Technically, dataframes are a specific type of matrix that provide more flexibility when calling/using data whereas matrices can be manipulated using common tools of linear algebra, like transpose (`t()`). Regardless of the differences, matrices and dataframes are easily transformed into one another using `as.data.frame()` and `as.matrix()`.

You can think of dataframes as organizing different variables (column vectors) by observations (row vectors). Specifically, a row denotes an observation/individual whereas a column denotes a variable. The intersection of the two, let's say row 3, column 2 (notated `[3,2]`), is the value of variable 2 for observation 3 (e.g. Individual 3 has an income of $30,000). So we can either look at the dataframe by column (variable) or row (individual).

Often you will import entire dataframes for analysis, but sometimes you may merge vectors together to create a dataframe:

```
Name <- c("Sam", "Liz", "Elon")
Income <- c(20000,1000000,1000000000)
Gender <- as.factor(c("Male", "Female", "Male"))
Academic <- c(T,T,F)
df <- cbind.data.frame(Name,Income,Gender,Academic)
df

   Name     Income  Gender  Academic
1  Sam      2e+04   Male    TRUE
2  Liz      1e+06   Female  TRUE
3  Elon     1e+09   Male    FALSE
```

The benefit of a dataframe is the organization of a lot of variables across a lot of observations into the same object. Furthermore, it displays the true structure of the data: many different observations with various and often unique combinations of variable values.

Let's say that we only want to access some portion of the data, what should we do then? For example, what if we only want to use the Income data from the dataframe or, even more complex, we only want the observations from the dataframe that have an income equal to or above $100,000? To do that, we can subset the dataframe:

```
df[1,2]         #Pull the income for Sam, first value is the row, second is the column.
df$Income       #Pull the entire Income vector using the $ and the name of the variable.
df[,2]          #Pull the entire Income vector using brackets, denoting the second column.
df[1,]          #Pull the entire row vector for Sam using brackets, denoting the first row.
df[,1:2]        #Pull the Name and Income vectors for every observation.

df$Income[df$Name == "Sam"] #Pull Sam's income using a variable subset.
df[df$Income >= 100000,]    #Subset the dataframe to individuals with income equal/above 100,000.
df[df$Academic == T,]       #Subset the dataframe to academics.
df[df$Academic == T & df$Gender == "Male",] #You can also subset based on multiple variables.
#Vectors are subset by just brackets [], dataframes are subset by row and column brackets [,].
#This is because there are no rows/columns, just a single string of values for vectors.

#You can also subset using the subset function, which behaves much the same way.
subset(df$Income, df$Name=="Sam")
subset(df, df$Income >= 100000)
subset(df, df$Academic == T & df$Gender == "Male")
```

Another important data class, but less often used with traditional data, is lists. The benefit of lists are that they can store vectors of varying lengths in the same object whereas matrices and dataframes require vectors to all be of the same length. An example:

```
K <- seq(1, 4, 0.5) #Produces every .5 number from 1 to 4 (i.e. 1,1.5,2,2.5,3,3.5,4).
L <- letters[1:5]   #Produces the first 5 letters (i.e. A,B,C,D,E)
M <- list(numbers=K, characters=L) #Stores K, length 7, and L, length 5,
                                    in the same object.
```

Outputs from various statistical models that you will run in R will often be lists, e.g. OLS results:

```
ols.results <- lm(Income ~ Academic + Gender, data = df) #Run a basic OLS on our data.
class(ols.results)                                       #Check the class of the object
```

Dataframes and vectors (and sometimes lists) comprise the primary framework for data-science in `R`: you will encounter these throughout your work. The next step is understanding how to manipulate and use data for different purposes. We already saw basic manipulations (e.g. addition, subtraction), but *functions* allow us to do much more than those simple operations including, but not limited to, Linear Regression (OLS), Maximum Likelihood Estimation (MLE), and Machine Learning.

1.2.4 Base Functions

`R` has a number of common functions already built into the program (others are user-contributed and are accessed by installing and loading libraries; more on that soon). For example we have basic math functions:

```
sqrt(100)                   #Take the square root of 100.
E <- mean(B)                #Assign E the mean value of B.
G <- median(B)              #Assign F the median value of B.

B <- rnorm(10000, mean = 100)   #Sample 10,000 times from a normal distribution centered at 100.
mean(B)                     #Take the mean of the vector B (should be close to 100).
hist(B)                     #Plot a histogram of the values of B.
```

While these are relatively simple examples, these functions should give you a taste of the capabilities of R. Note, many functions have default values for arguments if they are not explicitly defined in the function (e.g. `rnorm()` assumes a mean of 0 and a variance of 1 if not specified). Many complicated functions are actually "nests" of simpler functions. For example, the mean function can actually be coded by hand as:

```
Z <- c(1,2,6)
sum(Z)/NROW(Z)  #This is simply the sum of Z divided by the number of components in Z.

mean.function <- function(x) {  #One can make functions in R using this command.
    sum(x)/NROW(x)  #You can also use the function length() in the denominator.
}
mean.function(Z)

#An even more fundamental coding would be:
(Z[1]+Z[2]+Z[3])/3
```

The convenience of functions is that you don't have to code all of the components (unless you want to make a new function, which we will cover later). Instead, all you have to do is specify the arguments within the function, i.e. what object(s) and/or values the function is acting upon or using.

1.2.5 Random Numbers

We can generate (pseudo)random numbers[5] in R with the following code. For instance, let's roll a die:

```
sample(1:6, 1) #This "rolls" the die one time. In other words, it randomly draws once from 1-6.
sample(1:6, 100, replace=T) #This rolls the die 100 times.
#The replace argument essentially makes this a die,
whereas without replacement it would be like drawing 6 numbered slips of paper from a hat:
sample(1:6, 6, replace=F)

sample(1:20, 100, replace=T) #Now with a 20 sided die.
sample(1:6, 100, replace=T) + sample(1:6, 100, replace=TRUE) #This rolls and sums two dice.
```

Now let's draw a histogram of the values of rolling and summing two dice and look at the means/medians:

```
#Let's roll the dice 10 times and see how close to the expected value of 7 we get:
rolls1 <- sample(1:6, 10, replace=TRUE) + sample(1:6, 10, replace=TRUE)
hist(rolls1)
mean(rolls1)
median(rolls1)

#What happens when we increase the draws to 100?
rolls2 <- sample(1:6, 100, replace=TRUE) + sample(1:6, 100, replace=TRUE)
hist(rolls2)
mean(rolls2)
median(rolls2)

#What happens when we increase the draws to 10000?
rolls3 <- sample(1:6, 10000, replace=TRUE) + sample(1:6, 10000, replace=TRUE)
hist(rolls3)
mean(rolls3)
median(rolls3)
```

What phenomenon are we witnessing here?

[5] There are no truly random numbers in computer science, as random number generators (RNG) actually use a "seed" number to generate strings of numbers through an algorithm from which to sample from to get "random" numbers. However, for our purposes (and those of most others) these numbers are sufficiently random.

1.2.6 For() Loops and the apply() functions in R

Frequently, we want to use a function on numerous sets of values, like every row (observation) in a dataframe. Or we just want to run a function a set amount of times, like 10. For these we can use loops. While there are actually 3 kinds of loops (`for()`, `while()`, and `repeat()`), we will primarily focus on the `for()` loop as it is the most commonly used. So let's say we want to square all of the values in a vector, we could construct a `for()` loop that looks like:

```
V <- rnorm(100)      #Draw 100 random normally distributed values.
V2 <- NA             #Create a vector to hold the values.

for(i in 1:100) {    #For every i value in 1-100, do what is in the squigly brackets using i.
  V2[i] <- V[i]*V[i] #Multiply every ith value of V by itself and store it as the
                      ith value of V2.
}
```

While this is certainly intuitive, `for()` loops can actually be very inefficient because they have to do each iteration after the other, instead of making the calculations in parallel. Functions, however, can be "applied" to any number of objects (we'll focus on a vector here) and is much more efficient than loops because they can be applied in parallel.[6] We can code functions and assign them to objects like any other type of object, but we denote the amount of arguments, and what they're called, as well. Once we code the function, we use the specific `apply()` function that best matches out intended purpose and/or object that we are acting on to finish our calculations. So, let's code the previous loop as a function and use the `mapply()` function to calculate the squares:

```
squarey <- function(x){     #This function takes a single argument "x"
  x*x                       #This multiplies the value x by itself.
}

V2 <- mapply(squarey, V)    #This applies squarey to every value in V.
```

Functions can get much more complicated but will also be a lot more efficient than loops when used with larger dataframes. Keep this in mind when manipulating large dataframes and consider switching to a function if a loop is taking too long. Best practice is to always use functions and `apply()`, but often the difference in efficiency is so negligible that it is easier to code a simple loop. Furthermore, loops are arguably easier to understand intuitively than applying functions. There are also times that `while()` is useful and, to my knowledge, there is no analogous form of applying a function.

1.3 Matrices in R

While we already covered dataframes earlier, matrices and their manipulation are important in many of the functions used in R like `lm()` for OLS. This section is not incredibly important to understand at this point, but if you do want to understand the inner workings of many functions or if you want to code your own functions and packages, it will likely be necessary to understand these operations. We can create and manipulate matrices in R with the code:

[6] You won't notice significant differences for data and operations that are so small, but the more data and the more complex the operations, the larger the difference you will notice between applying functions and using a loop.

```
A <- matrix(c(1,3,0,-1,6,2), nrow=2, ncol=3, byrow=TRUE)
#This creates a matrix denoting the values and number of columns and rows.
#The argument "byrow" denotes how R takes the vector and transforms it into a matrix.
A
A[1,]   #Pulls the first row of A.
A[,1]   #Pulls the first column of A.
A[,1:2] #Pulls the first and second columns of A.
A[2,3]  #Pulls the value of column 3 for row 2.
```

1.3.1 Matrix Addition and Subtraction

```
B <- matrix(c(3,1,2,0,5,1), nrow=2, ncol=3, byrow=TRUE)
A + B
A - B
```

Remember, only matrices of the same exact size $dimension(A) = m \times n = dimension(B)$ can be added or subtracted from one another.

1.3.2 Matrix Multiplication

```
A
2 * A #Multiplies all values in the matrix by 2.
D <- matrix(c(1,-2,0,5,4,3), nrow=3, ncol=2, byrow=TRUE)
E <- matrix(c(3,1,4,-1,2,5), nrow=2, ncol=3, byrow=TRUE)
D%*%E #Gives you a 3x3 matrix.
E%*%D #Does not work.
```

Remember, matrix multiplication is almost always *not* communicative (i.e. $DE \neq ED$) and requires the number of columns of the left matrix to equal the number of rows in the right matrix. This will also result in a matrix that has the same number of rows as the left matrix and columns of the right matrix.

1.3.3 Matrix Trace and Determinant

```
H <- matrix(c(5,-3,-5,10), nrow=2, ncol=2, byrow=TRUE)
sum(diag(H))    #Sums the main diagonal values of the matrix H.
det(H)          #Takes the determinant of H.
```

The determinant of a non-invertible matrix is zero, whereas it is nonzero if it is invertible. Also, only $n \times n$ matrices can be invertible, but not all are.

1.3.4 Matrix Transposition and Inversion

```
J <- matrix(c(1,2,3,4,5,6,1,8,9), nrow=3, ncol=3, byrow=TRUE)
t(J)        #Transposes J
solve(J)    #Finds the inverse of J.
```

A matrix has elements $[i, j]$ equal to the $[j, i]$ elements of the transpose of that matrix.

In linear algebra, the Identity matrix (an $n \times n$ square matrix with only 1s on the diagonal and 0s elsewhere) acts in the same way as the number 1 does in regular algebra. Also, remember that the inverse of a number (e.g $inverse(2) = 2^{-1} = \frac{1}{2}$) times a number equals 1 (e.g. $\frac{1}{2} \times 2 = 1$). So, the inverse of a matrix (A^{-1}) times the original matrix (A) equals the $n \times n$ Identity matrix (I) (i.e. $A^{-1}A = AA^{-1} = I$).[7]

1.4 Installing and Loading R Packages

R is open-source, meaning users can contribute packages that are made available to all R users via CRAN. You'll need to both install and load all of these packages:

```
install.packages(c("BMS","caret","ClassDiscovery","corrplot","datasets",
"doParallel","deepboost","dplyr","extraTrees","fastICA","foreach","foreign","gbm",
"GenAlgo","ggfortify","ggplot2","kernlab","lavaan","MASS","MCMCpack",
"mlbench","nnet","pROC","quadprog","randomForest","RANN", "readit"))
library(deepboost)  #Load one package.
help(deepboost)     #Open the documentation for that package.
```

Packages are installed using the `install.packages()` function with the names of the desired packages in quotation marks, and concatenated if installing more than one. Once packages are installed, to use them you must load them using the `library()` command (no quotation marks this time).

1.4.1 Loading Data from Packages

We can load data from a package such as `car` package by loading that package and then using the command:[8]

```
install.packages("car", dependencies=TRUE)
library(car)
data(States) #Loads the States dataset from car.
```

To view the first few observations, use:

```
head(States)
```

Here are some examples of commands and code that we've already learned using a real dataset:

We can access a single variable using, for example:

```
States$SATM
```

Suppose that we wanted to only examine states in our dataset with an average SATM score of more than 500. We could create this subset by typing:

[7]This is one of the few cases where matrix multiplication is communicative. It does not matter whether the matrix is left or right multiplied by its inverse, it will equal the identity matrix either way.

[8]Note: Loading data from packages is usually only for using test data to try the package out. Loading regular datasets is done in a different manner.

```
States.highSATM <- subset(States, subset = SATM > 500)
```

We could count the number of states with average SATM scores of greater than 500 using (at least two) approaches. One is:

```
dim(States.highSATM)
```

Another way is to count the number of times a condition is met:

```
States$SATM > 500
table(States$SATM > 500)
```

1.4.2 Loading Data from Outside Files

There are many different packages used to load non-`R` files, however, a relatively new package called `readit` solves this problem and can open all of the commonly used data files. To open a dataset, you either have to set what's called your working directory (where R is looking for files) or you have to specify the full file path in the function.[9] Examples of both cases (using Windows and my username) are below:

```
library(readit)
#Loads a .dta (Stata) file using the full path.
legis.dat <- readit("ANES2016.dta")
#Loads a .txt (text) file using the full path.
legis.txt <- readit("bes_2010.txt")
#Loads a .csv (comma delimited) file using the full path.
legis.csv <- readit("incumbent.csv")
#This is to load a saved R object.
data.final <- readRDS("data.final.rds")


setwd("E:/Dropbox/ICPSR_2020/labs/data") #Set working directory.
legis.dat <- readit("ANES2016.dta")     #Loads a .dta (Stata) file.
legis.txt <- readit("bes_2010.txt")     #Loads a .txt (text) file.
legis.csv <- readit("incumbent.csv")    #Loads a .csv (comma delimited) file.
data.final <- readRDS("data.final.rds") #This is to load a saved R object.
```

Important to note too, is to name your file paths without spaces for any folders that you use for datasets, instead use underscores (_) or dashes (-). Finally, `readit` is lovely and I highly recommend it for loading any non-`R` data files.

[9] If you have a Mac, you can just copy and paste the file path, if you have Windows, you can copy the path, but you have to change each backslash (\) to a forward slash (/).

1.5 Project Euler

While not specifically related to data science, `projecteuler.net` has many coding problems/puzzles that help you think through how `R` works in more fundamental ways. One of my favorite problems that uses subsetting and `for()` loops is:

Here's a 20x20 grid (matrix) of numbers:

```
08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48
```

What is the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the 20×20 grid? How might you go about tackling this problem? Feel free to actually complete it if you want!

An extra for fun, and is a little more complicated:

The four adjacent digits in the 1000-digit number that have the greatest product are $9 \times 9 \times 8 \times 9 = 5832$.

```
7316717653133062491922511967442657474235534919493496983520312774506326239578318016984801869
4788518438586156078911294949545950173795833195285320880551112540698747158523863050715693290
9632952274430435576689664895044524452316173185640309871112172238311362229893423380308135336
2766142828064444866452387493035890729629049156044077239071381051585930796086670172427121883
9987979087922749219016997208880937766572733300105336788122023542180975125454059475224352584
9077116705560136048395864467063244157221553975369781797784617406495514929086256932197846862
2482839722413756570560574902614079729686524145351004748216637048440319989000889524345065854
1227588666881164271714799244429282308634656748139191231628245861786645835912456652947654568
2848912883142607690042242190226710556263211111093705442175069416589604080719840385096245544
4362981230987879927244284909188845801561660979191338754992005240636899125607176060588611646
710940507754100225698315520005593572972571636269561882670428252483600823257530420752963450
```

Find the thirteen adjacent digits in the 1000-digit number that have the greatest product. What is the value of this product?

What would be some possible ways of answering this problem? (Hint: my solution exploits zeros and how strings can separated by a given character, e.g. "2", and the fact that lists can store objects of different lengths.)