Beginner R Programming for Anthropologists (and other folks as well)

Allison E. Mann, July 24, 2016

What is R and why should you use it?

R is a statistical and graphics package (NOT a programming language, though there are some similarities). Benefits of using R over boutique statistical analysis software:

- **R is freely available**
- Because R is open-source there are many packages that can be added for nearly any application
- Publication ready, flexible graphics capabilities
- Supportive, active community
- Widely used and respected platform for statistical analysis

Like many programming languages (e.g. python, Java, C++) R is based on an object-oriented language (called S) meaning that all basic units in R are known as “objects”. In other words, an object in R is ANY INPUT (e.g. numbers, letters, logical arguments, etc).

Installing R

- Windows: https://cran.r-project.org/doc/manuals/r-release/R-admin.html#Installing-R-under-Windows
- Linux: https://cran.r-project.org/doc/manuals/r-release/R-admin.html#Installing-R-under-Unix_002dalikes
- Mac OS: https://cran.r-project.org/doc/manuals/r-release/R-admin.html#Installing-R-under-OS-X

Running R

There are a variety of GUI options for running R. Among the most popular is RStudio (https://www.rstudio.com/). However, you can also run R directly from the terminal. To open up an R workspace open a terminal (or command line) and type R at the prompt. You should see a similar message to the one below if R is installed correctly. The > or carrot symbol is your R prompt. All commands are typed at the R prompt in this tutorial.

```
R version 3.3.0 (2016-05-03) -- "Supposedly Educational"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

For fun type in `> demo(graphics)` to see some of the graphing capabilities of default R!

More R resources

- Official R homepage: https://www.r-project.org/
- R for beginners: https://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf
- Code School, Try R: http://tryr.codeschool.com/

Some important definitions in R:

- *Vector* = a one dimensional array or collection of objects that are **all of the same type**
- *Function* = a set of instructions written in the R syntax that are to be carried out on an object of elements of an object
- Argument = user provided values to be used in a function

- *Parameter* = a variable that stores arguments passed to your function (similar to argument, sometimes used interchangeably)
- *Operator* = a symbol that indicates some meaning or relationship among objects
- *String* = a character or characters (e.g. words, sentences, etc)
- *Int* = short for integer – whole numbers
- *Float* = floating point numbers, so while 4.0 is a float, 4 is an int

Types of operators (separated by commas, not a complete list):

General operators:

• `<-, ->`	#assigns something to an object
• `~`	#assigns relationships between objects, often in formulas
• `:`	#assigns some sequence (usually of numbers)
• `$`	#indicates that object on right of operator is component of object on left of operator (e.g. a column from a dataframe
• `“” or ''`	#object is a string or single character
• `#`	#comment

Math operators: `+, -, *, /, ^, =`

Comparison operators: `<, >, >=, <=, ==, !=`

Boolean operators: `!, &, && (and if), |, || (or if)`

Five commandments for naming objects in R:

1. Objects MUST begin with a letter – this is good practice in any case, never give sample IDs or other variables in your datasets numerical names – there may be programs that treat them in unexpected ways
2. Cannot contain operators or other special characters that R uses – so no: `, - = * / # % & [ ] { } spaces`
3. Be specific but brief! You should be able to know what they are from the name itself – 20 minutes from now you might not know what object `X` contains but the variable name `length.of.the.final.iteration.of.this.analysis` is cumbersome to type
4. Objects are case sensitive! `Df1` is the the same as `df1`
5. If you must, separate words in your object names with either “.” or “_”. So: `toe_length` or `toe.length` OR use “camel case” e.g. `toeLength`. All are equally valid as object names – the differences come down to aesthetics.

R command line expressions:

The basic structure of a R command is the **command prompt** followed by an **object**, the **assignment operator**, and some **expression**. Once you assign an expression or some other function/data to an object you can call that object to evaluate your expression. So say we have a small dataset of big toe length and we want to assign an object called toe.length to a vector that has all of our observations.

```
> toe.length <- c(3,5,4,3,6,7,3,3,2.5)
```

The “c” outside of the parentheses tells R that we want to concatenate these values into a new object called toe.length. We can then evaluate the expression by typing in the name of the object:

```
> toe.length
[1] 3.0 5.0 4.0 3.0 6.0 7.0 3.0 3.0 2.5
```

You'll notice that R did something unexpected here. You'll remember that vectors must be a collection of objects of all the same type. Since you included a floating point element it automatically converted them all to floats. Be aware of this!! Floats are often not mathematically treated the same as integers.

We can now use some of R's build in functions to further evaluate our new object. So for example if we wanted to know the mean toe length:

```
> mean(toe.length)
[1] 4.055556
```

Or maybe we want to know how long our vector is (this is really useful for loops!)

```
> length(toe.length)
[1] 9
```

Classes of objects:

Knowing the class of your object is important as R will treat different classes of objects differently.

1. *Numeric* → integers, floats
2. *Characters* → strings, letters, numbers

3. *Boolean* → TRUE, FALSE
4. *Vector* → a single dimension of some type of information
5. *Matrix* → a multidimensional collection of objects (of same kind)
6. *Data Frame* → multidimensional collection of objects (with each column representing different variables, can be objects of different kinds)
7. *List* → collection of objects that have different classes (e.g. can be a collection of numbers and strings)
8. *Factor* → categorical (factorial) class (e.g. if you have a variable that is Male and Female, factor of two levels)

We can check the class of an object in our workspace by running the class() function:

```
> class(toe.length)

[1] "numeric"
```

To get a little more information on the structure of our object we can run the str() function:

```
> str(toe.length)

 num [1:9] 3 5 4 3 6 7 3 3 2.5
```

This command tells us that our object's class is numeric, it has 9 total elements as well as the object's values as originally designated.

What if you want to check if an object is of a particular type?

```
> is.numeric(toe.length)

[1] TRUE

> is.character(toe.length)

[1] FALSE
```

You can also coerce your object into a different class of objects (note that because we do not assign this expression to a new object nothing changes to our old object):

```
> as.character(toe.length)

[1] "3"   "5"   "4"   "3"   "6"   "7"   "3"   "3"   "2.5"
```

Keeping track of your R workspace:

```
> ls()          #list the objects currently in your workspace
> rm()          #remove the object name within the brackets
> getwd()       #see the current working directory
> setwd()       #set a new working directory
> dir()         #see all files currently in your working directory
```

Installing and loading new packages in R:

```
> install.packages()
```

This will bring up a list of repositories that you can choose from to download from. Once you choose a repository, a list of all available packages will load. If you already know the name of the packages you want you can download them by typing in the name of the package surrounded by quotes in the install.packages function parentheses. Install some packages.

```
> install.packages(“reshape”)

> install.packages(“lattice”)
```

To load these packages you'll need to call them directly with the library() function (notice that you don't need to include quotation marks):

```
> library(reshape)
```

Getting help

If you are not sure how to use a particular package you have a couple options. If you know the exact package name you can run the help function to open the library's documentation page (click q to quit):

```
> help(reshape)
```

```
OR

>?reshape
```

If you are interested in a specific built in functions of a package you can type:

```
> ??reshape::melt
```

Which gives you some basic information on the different melting options (converting wide-format to long-format data) in the reshape package. You can then get some more information on these with the help command.

```
> help(melt_check)
```

Finally, if you don't know what package or function will work for you, remember that Google is your best friend. Use the appropriate R terminology to help you track down what you are looking for!

Loading your own data into R:

Some things to remember – R is fairly particular about how it wants your data to be set up for it to load properly. First and foremost your data should have NO EMPTY SPACES. R does not take kindly to empty cells or null values. If you need to have null values set them up as “NaN” or “NA” so R will recognize them. Never assume that R has loaded your data in properly, even if there was no load error. Always check that your values were loaded in the expected fashion.

If your data is separated by some common unit it is simple to load your data using the read.csv function. Let's make a dummy file and load it into R. First use the system function to call a shell script:

```
> system(“nano file.txt”)
```

Create the following file with tab delimitation. Save to file by pressing CTRL+X:

```
sample    name length
samp1     moxie     24
samp2     laika     22
samp3     jackson   30
```

We can now read in the file into R:

```
> data <- read.csv(“file.txt”, header=TRUE, sep=”\t”)
```

In this example you are reading in a file in your current working directory called file.txt, telling R that the first row should be treated as column headers, and that your file is tab delimited.

A brief primer on indexing:

Indexing is a way that you can access slices or particular elements of your data using the [ ] operators. As vectors are single dimension arrays their indexes are more simple than matrices or data frames. So for example, if we wanted to access the 3rd element of our toe.length vector we could do so like this:

```
> toe.length[3]

[1] 4
```

What if you wanted a known subset of your data?

```
> toe.length[1:4]

[1] 3 5 4 3
```

Remove or ignore a value in your vector (why do you think this is formatted as floats while the others are ints?)?

```
> toe.length[-4]

[1] 3.0 5.0 4.0 6.0 7.0 3.0 3.0 2.5
```

Indexing multidimensional arrays is a little more complicated but will be easier if you remember that row is always called first followed by column.

AXIS 0↓**:** AXIS 1→

	1	2	3
1	[1,1]	[1,2]	[1,3]
2	[2,1]	[2,2]	[2,3]
3	[3,1]	[3,2]	[3,3]

NOTE: While most programming languages start indexing at 0 for the first element of some array, R starts with 1.

So if I wanted to access the middle value here I could do so a couple ways. First let's transform our vector into a proper matrix:

```
> mat <- matrix(toe.length, nrow=3, ncol=3)
```

How did R fill in our matrix? By columns or rows? Any interesting or unexpected outcomes? Why do you think this is?

```
     [,1] [,2] [,3]
[1,]    3    3  3.0
[2,]    5    6  3.0
[3,]    4    7  2.5
```

Let's say you want to access whatever value is in the second column in the third row. You could do this like so:

```
> mat[3,2]
[1] 7
```

If, on the other hand, you had column names you could use them to access values in a more intuitive way. Let's say our matrix is actually looking at toe length measured at different seasons. We can assign column names like so:

```
> colnames(mat) <- c("spring", "summer", "winter")
```

If each row represented a different sample we could also change this like so:

```
> rownames(mat) <- c("sample1", "sample2", "sample3")
```

And then convert our matrix into a more friendly data frame (matrices are typically for data of all the same type while data frames expect your columns to all be variables of different types):

```
> df <- as.data.frame(mat)
```

And then access elements in a column or whole column of values by using the component operator.

```
> df$spring

[1] 3 5 4

> df$spring[3]

[1] 4
```

Loops and Functions in R

There are two commonly used types of loops in R. The for loop and the while loop. Loops ask R to do some function multiple times according the the parameters you provide it. Loops, like functions (below) are bounded by curly brackets that indicate the beginning and end of the loop.

For loop example:

First let's initialize some values.

```
> str <- "blast off!"
```

We want each element of this string so we can split it with string split (essentially what this is doing is splitting each element of our string since we've split it by no spaces. You can set this to any delimiter you want.)

```
> split <- strsplit(str, '')[[1]]

> iter <- 1
```

Now let's use this to create our for loop

```
> for(i in 1:length(split)){

countdown <- paste(i, " ", split[iter])

print(countdown)

iter <- iter+1

}
```

While loops are very similar except that they evaluate whether or not a statement is true in each iteration (hint: zero is considered FALSE in boolean logic). We can write a similar while loop as our for loop with some minor modifications (remember to reset any iteration objects!).

```
> iter <- 1

> countdown <- length(split)

> while(countdown != 0){

blastoff <- paste(countdown, " ", split[iter])

iter <- iter + 1

countdown <- countdown - 1

print(blastoff)

}
```

Sometimes you might find that no package or function currently exists to meet your needs in R. In that case you can write your own functions! The basic syntax for functions in R is as follows where the open and closed brackets indicate the start and end of the function (remember to assign your function to an object name!:

```
> functionName <- function(<arguments that will need to be passed to the function>){
```

```
<evaluation statements>

}
```

Let's make a very simple counter function that uses a loop

```
> myFunc <- function(num){         #Start the function

vec <- 0                      #Initialize the value of vec to zero

for(i in 1:num){              #Start our loop, will run from one to the value of num

vec <- c(vec, i)              #After each iteration, vec adds an element of value i

}

return(vec)                   #Have function return vec outside of loop.

}                             #Close function
```

To run this we can call it like a normal function and include any desired value for num

```
> myFunc(20)

 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Let's try to understand what is happening under the hood of this function a little better. We open the function by assigning our starting number (vec) as zero, after which it begins a loop that will iterate over each element (i) in a series of numbers from 1 to the value of num. If we run this function with num=5 the expression:

```
vec <- c(vec, i)
```

Is actually run five times. Each time vec is concatenated with the next value of i. So for example in pass one:

```
vec = 0 + 1
```

pass two:

```
vec = 0, 1 + 2
```

pass three:

```
vec = 0, 1, 2 + 3
```

pass four:

```
vec = 0, 1, 2, 3 + 4
```

pass five (final pass):

```
vec = 0, 1, 2, 3, 4 + 5
```

When this is returned you then have a full vector of six numbers ranging from zero to five.

Working with the Howells craniometric dataset:

Alright so let's put this all into practice with a real dataset. In the zip file you downloaded from git hub should be Howells_dat.csv. Craniometric data from this file was collected by William Howells from 1965 to 1980 and includes measurements from more than two thousand individual crania from 28 populations of varying ages (download the original data here: web.utk.edu/~auerbach/HOWL.htm). Howells detailed the data collection in his 1996 *American Journal of Physical Anthropology* publication “Howells Craniometric Data on the Internet”. One of the major benefits of this data is that Howells took all of the measurements himself, reducing worries of inter-observer bias. In addition, I've added some factor based climate data calculated from purported origin of the cranial material itself as classified by the Köppen-Geiger climate classification system (*Peel et al. 2007. Updated world map of the Köppen-Geiger climate classification. Hydrol. Earth Syst. Sci*). The impact of climate on craniofacial shape is a classic discussion of anthropology hypotheses of which we can test with rich datasets such as this one. On the other hand, large datasets can be difficult to explore using boutique or less flexible statistical software. R to the rescue!!

Step One: Take a peek at and load in the data

Make sure you are in the correct folder

```
> dir()
```

Take a look at your file, read in and attach

```
> system("head Howell_dat.csv")

> crania <- read.csv("Howell_dat.csv", header=TRUE, sep=",")

> attach(crania)
```

Step Two: Take a look at the data attributes

Let's see what column names we have

```
> names(crania)
```

Summary statistics for the full dataset?

```
> summary(crania)
```

How many males and females do we have? What climate variables are present? What populations do we have?

Step Three: Any notable differences between populations (separated by sex)? Let's make a boxplot that shows this for one measurement.

```
> library(lattice)

> bwplot(GOL ~ Sex | Population, xlab="Sex", ylab="GOL")   #Note: if you don't have the data
attached, you'll need to call the data frame with the component operator ($)
```

Any notable problems?? For some populations we don't have any females!!

Step Four: Split your data by sex, detach full dataset, attach just your males

```
> males <- crania[Sex == "M",]

> detach(crania)
```

Step Five: Let's look at a measurement by climate groups

```
> bwplot(males$GOL ~ males$climate_group)

> hist(males$GOL, prob=TRUE)
> lines(density(males$GOL))
```

Step Six: What if you wanted to look at all of the measurements by climate groups and save as a pdf? With a loop of course!

First let's create a vector with just our measurement names that we need

```
> names(males)

> measurement.list <- names(males)[10:79]
```

How many variables do we actually have?

```
> str(measurement.list)
```

Let's make an appropriate path for your files

```
> system(“mkdir boxplots”)
```

Now we can start the loop!

```
> for(i in 1:70){

id <- measurement.list[i]

file <- paste(id, “_climatebp.pdf”, sep=””)

path <- paste(“boxplots/”, file, sep=””)

pdf(path)

print(bwplot(males[,id] ~ males$climate_group, ylab=id, main=paste(id, “by Climate Group”,
sep=” “)))

dev.off()

}
```