OEB 153 – Lab 0

09/26/2014

Welcome!

Welcome to R! R is a statistical computing environment. This means it is many things. R is

- a program for data analysis and exploration,
- a fancy calculator,
- and a fully fledged programming language.

In all these uses, R is oriented towards statistics. There are multiple ways to use R, and for your own statistical projects, you'll probably use R in a variety of ways. We'll start with the basics.

R is also open source. This means that every bit of R's source code is visible to anyone that wants to look at it. This also means that R is free. Most importantly, R being open source means that R users can contribute their own code ("packages" or "libraries") for others to use. This particular feature has made R the *de facto* standard for statistical computing. In a variety of fields, it is common for researchers to contribute R packages whenever they introduce new statistical tests or procedures.

Parts of this introduction are adapted from the introduction to R for the Whitlock and Schluter textbook. Check it out!

As we work through this introductory Lab, it will be tempting to copy and paste commands from this document into R. Don't!

Getting Started

Opening R

The most basic way to use R is to type commands into an R terminal. There are several ways of opening an R terminal. You can

1. open Terminal ("Command Prompt" in Windows terminology) and enter the command `R`
2. double click on the "R" icon (or open the R program in Windows), opening a graphical interface with an R terminal
3. use the console (synonymous with "terminal") in the lower left corner of the RStudio screen layout

Regardless of which way you choose to interact with R, you should see a prompt that looks like a `>`. This is R waiting for your input.

```
>
```

Type `3+4` into the prompt and hit Return. You should see something that looks like this:

```
> 3+4
```

```
[1] 7
```

Congratulations, you've just calculated something in R! The `[1]` in the output just signifies that what was printed to the screen is the first (and in this case, only) part of the output of a command.

R as a calculator

Using R as a calculator like this is actually pretty handy. This may seem far removed from doing statistical analysis in R, but working with R as a calculator is the best way to learn the basics. Try the following commands.

```
> 3+4*7
> 21.34/3.8 - 12.9
> (3+4)*7
> (1+3)^2
```

Notice that R respects order of operations. Also, a note on spacing: the expressions `(3+4)*7` and `(3+4) * 7` are completely equivalent in R. Spacing tends not to matter very much in R and is left up to the user as a matter of taste and readability.

If you forget to close a parenthesis, R is unforgiving. For example, if you enter `6+(14*9` into the R terminal, you'll see the following:

```
> 6+(14*9
```

```
+
```

The `+` sign is R's way of telling you that it needs more input to finish a command. Here, it's looking for a closing parenthesis. If you failed to notice that you were missing a closing parenthesis and went on to try to calculate `741/35.6`, you would get an error message:

```
> 6+(14*7
+ 741/35.6
Error: unexpected numeric constant in:
"6+(14*7
741"
```

If you ever find yourself with a `+` prompt, a convenient way to get your `>` prompt back is to enter something like `)))))))))))))`, forcing an error and returning you to the regular `>` prompt.

Functions

R has many useful built-in functions that you can call to perform calculations on your data. For example, the `log()` function calculates a logarithm.

```
> log(47.2)
```

```
[1] 3.854
```

Here, `log()` is the function, and `47.2` is the value we provide as an argument to `log()`. In general, we call a function by typing the function name and surrounding some number of arguments in parentheses.

By default, `log()` returns the natural logarithm, base $e = 2.7183$. Often functions have multiple arguments that can be used to control their behavior. For example, you can specify an optional `base` argument to `log()` to have the function return a logarithm of a different base:

```
> log(16,base=2)
```

```
[1] 4
```

If you know the assumed ordering of the arguments, you can provide both arguments without having to specify that the second is the base.

```
> log(100,10)
```

If you specify the name of every argument, you can provide them in any order you want.

```
> log(base=10,x=100)
```

Of course, you can also write your own custom functions to carry out just about any calculation you can imagine. We'll get to that in a later in the course.

Help!

How does one learn the arguments to a function, their names and assumed order? How does one know whether the logarithm function is `log()` or `logarithm()`?

R provides a help page for every built-in function. If you know the name of the function, you can enter `help(functionname)` or equivalently `?functionname` into the terminal, where you replace `functionname` with the name of the function you want to learn more about. Try typing `?log` into the terminal. You should see a page that gives a verbal description of `log()` (and related functions), examples of how to use `log()`, and a detailed description of the arguments the function expects. These help pages are always accurate and exhaustive, but they may be difficult to interpret at first.

If you didn't know that `log()` is the logarithm function, you can search for functions with "logarithm" in their description by typing the command `??logarithm`. You should see the names of a few functions and their descriptions. (The `base::log` notation means that `log()` is in the standard library `base`, which doesn't need to be loaded specially.)

The `??function` way of getting help is not always helpful. For example, you might expect that `??cosine` would help you find the cosine function, `cos()`. But this is not the case. Instead, here, Google is your friend. If you search Google for "cosine in R", you'll find the help page for the trigonometric functions in R, including `cos()`.

In general, if you are having trouble with something in R, try Google. There are many sites, email lists, and forums on the internet dedicated to helping people with R. Searching for a particular phrase can be especially helpful with error messages, which can be difficult to parse as a newcomer to R.

Variables and assignment

In this section we begin to use R more like a programming language and less like a calculator. One of the fundamental features of most programming languages is that values can be stored in variables. Rather than retyping your data again and again, you can refer to different aspects of your data by name.

Enter the following into your R terminal:

```
> y = 5
> x <- 4
```

Here we've created two new variables, `x` and `y`, and assigned the value `5` to the variable `y` and the value `4` to the variable `x`. In R, you can assign a value to a variable using either `=` or `<-` (an arrow, "less than" followed by "hyphen"). They are completely equivalent. In both cases, the value to the right of the operator is assigned to the variable to the left of the operator. You can name variables almost anything you want, and usually you want to pick something that is descriptive of what the variable represents. Try the following commands:

```
> x
> y
> x*y
> y = 7
> x*y
```

Note that variable names are case-sensitive. Try to calculate the product of `x` and (upper-case) `Y`, and notice that R produces an error.

Another thing to know: It is necessary to be explicit about mathematical operations you want R to do. For example, you might want to type the equation $z = 3x(1+4)$ as `z = 3x(1+4)`, but R gives an error. Instead, you'll need to enter all of the asterisks representing multiplication.

```
> z = 3*x*(1+4)
```

If you ever want to see what variables you have assigned in the current R session, use the command `ls()`. `ls` is short for "list."

```
> ls()
```

```
[1] "x" "y"
```

Vectors

So far we have made R do calculations with individual numbers, which is like doing statistics with a sample size of one. R is best when it deals with multiple data points at once. Enter the following in your terminal:

```
> data1 = c(6,5,1)
> data2 = c(4,3,2)
> data1
> data2
> data1+data2
> data1*data2
> log(data2)
```

Here we've used the `c()` function to create two *vectors* called `data1` and `data2`. (The `c` stands for "combine.") Notice how the `+` and `*` operators work here. All of the basic mathematical operators perform *elementwise* operations on vectors. Notice also that when we apply the function `log()` to the vector `data2`, it returns the natural logarithm of every element in `data2`.

There are many ways to create vectors besides typing out all the elements and enclosing them in `c()`. The `seq()` function creates sequences of evenly spaced elements. There are many ways to specify which sequence you want; see `help(seq)` for an explanation. A few examples to try:

```
> seq(1,10)
> seq(1,10,by=2)
> seq(1,10,length=7)
```

The function `rep()` creates vectors of repeated elements. Again, there are a variety of ways of using `rep()`; see `help(rep)`. A few examples:

```
> rep(7,4)
```

```
[1] 7 7 7 7
```

```
> rep(c(1,2),times=2)
```

```
[1] 1 2 1 2
```

```
> rep(c(1,2),each=2)
```

```
[1] 1 1 2 2
```

Random number generation

One of the most useful functionalities of R is random number generation. R makes it very easy to simulate random numbers. Try the following:

```
> normalData = rnorm(10,mean=0,sd=1)
> normalData
```

```
 [1] -1.2386 -1.7014  0.3650 -0.1637  0.1375 -0.8538 -0.6779 -0.7286
 [9]  0.5020  1.5217
```

```
> binomialData = rbinom(10,n=8,p=0.5)
> binomialData
```

```
[1] 5 9 6 4 5 4 8 3
```

In your terminal, you'll see numbers that are different than what's on this worksheet: They're random! Try it again and you'll get different numbers. We used `rnorm()` to simulate 10 independent and identically distributed (i.i.d.) random variables from a Normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$. Similarly, using `rbinom()` we sampled 10 i.i.d. Binomial random variables with the number of trials being $n = 8$ and the probability of success of each trial being $p = 0.5$. R has random number generators for a large number of named distributions. For all the distributions that have built-in simulators in R, see `?Distributions`.

Analyzing Vectors

A lot of doing statistics is performing calculations on collections of data. In R, this means a lot of time is spent analyzing vectors. There are a huge number of built-in functions that perform various calculations on vectors. This is maybe the majority of what R is. Try out the following functions, which summarize our sampled data in various ways:

```
> length(normalData)
> max(normalData)
> min(normalData)
>
> sum(binomialData)
> sum(binomialData)/length(binomialData)
> mean(binomialData)
> median(binomialData)
> var(binomialData)
> sqrt(var(binomialData))
> sd(binomialData)
> summary(binomialData)
```

Indexing

Sometimes in statistics you need to perform calculations on data points that match only certain criteria. To select different subsets of the elements of a vector, we can use R's indexing features. Try the following:

```
> x = c(9,5,3,8,4)
> x
> x[2]
> x[4]
> x[c(1,2,5)]       # first, second, fifth elements
> x[1:3]            # recall that 1:3 is short
> x[-3]             # without the third element
> y = c(1,1,5)      # store some indices in another vector y
> x[y]              # use y to index x
```

`x[2]` produces a vector of length 1, containing the second element in `x`. `x[1:3]` produces a vector containing the first three elements in `x`. (Recall that `1:3` is shorthand for `seq(1,3)`.) Similarly, `x[c(1,2,5)]` produces the first, second, and fifth elements in `x`.

R uses one-based indexing, meaning that the first element of `x` is `x[1]` and the second is `x[2]`. Some programming languages use zero-based indexing, meaning that the first element is `x[0]` and the second is `x[1]`. If you're used to programming in a language with zero-based indexing, watch out for this in R.

In the last two lines of code above, I added a comment to the code as an explanation of what was going on with the indexing. R ignores anything after a `#` in any command, allowing you to write comments to yourself. This is not very useful when we are just entering commands into the terminal, but later when we get to writing scripts, comments become crucial for making the code understandable.

Logical tests

Try the command `x < 5`:

```
> x
```

```
[1] 9 5 3 8 4
```

```
> x < 5
```

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

A logical variable (or just “logical”) indicates whether something is true or false. `TRUE` and `FALSE` (case-sensitive) are the two values that a “logical” variable can take in R. We could create a logical vector using the vector function `c()` like `c(TRUE, FALSE, TRUE)`, but in practice logical vectors are almost created using **logical tests**. In `x < 5` above, we applied the logical operator `<`, “less than,” to test whether the elements of `x` were less than 5. There are a handful of logical operators, each worth knowing about:

```
> x < 5             # less than
> x <= 5            # less than or equal to
> x > 5             # greater than
> x == 5            # is equal to (notice, TWO equals signs!)
> x != 5            # is not equal to
> x > 3 & x <= 8    # x is greater than 3 AND x is less than or equal to 8.
> x > 3 | x <= 8    # x is greater than 3 OR x is less than or equal to 8.
```

Logical vectors are usually most helpful in indexing, where they are used to create filters of data. For example:

```
> x
```

```
[1] 9 5 3 8 4
```

```
> x[x<5]          # a vector of all the elements in x that are less than 5
```

```
[1] 3 4
```

Logical vectors can also be good for counting. When `sum()` is applied to a logical vector, the `TRUE` values are treated as 1, and the `FALSE` values are treated as 0. So `sum(logicalVec)` counts the number of `TRUE` entries in the logical vector `logicalVec`.

```
> sum(x<5)
```

```
[1] 2
```

Data frames: working with multiple variables

At this point, we are very close to being able to work with realistic data. Usually, when we’re working with real data, there are multiple variables to consider. It would be okay to store each variable in a separate vector, but with more than a handful of variables, it becomes difficult to keep track of all the separate vectors. This is where it is useful to use a data frame.

In R, a **data frame** is a collection of vectors that each represent some variable describing aspects of data. We can create a data frame with simulated wine harvest dates using the function `data.frame()`:

```
> years = 1300:2013     # a vector containing the years of grape harvest
> numYears = length(years)
> harvestDate = round(rnorm(numYears, mean = 25, sd = 4))    # simulate harvest dates
> simHarvest = data.frame(years, harvestDate) # create data frame!
```

In general, create a data frame by calling `data.frame(var1, var2, var3, ...)`, where `var1`, `var2`, etc. are the variables you want to be in the data frame.

To get a sense of what a data frame is like, we can use the function `head()`, which displays the first few values of each variable in a data frame.

```
> head(simHarvest)
```

```
  years harvestDate
1  1300          31
2  1301          26
3  1302          26
4  1303          24
5  1304          24
6  1305          25
```

The `dim()` function is also useful. It returns the number of entries and variables in a data frame.

To interact with individual variables in data frames, we can use the `$` symbol. To refer to variable (i.e., vector) `var1` in data frame `dframe`, we would type `dframe$var1`.

```
> mean(simHarvest$harvestDate)
```

```
[1] 25.32
```

Here are a few things we can do with our simulated two-variable dataset:

```
> # calculate the mean harvest date
> mean(simHarvest$harvestDate)
```

```
[1] 25.32
```

```
> # calculate the max harvest date before 1500
> max(simHarvest$harvestDate[simHarvest$year < 1500])
```

```
[1] 34
```

```
> # count the number of harvest dates more than 28 days after Sep 1
> sum(simHarvest$harvestDate > 28)
```

```
[1] 146
```

Working with real data

Working from R scripts

Now we're ready to work with real data. So far we've just been typing commands into an R terminal. In most real-world uses of R, it is more practical to write commands into a **script** file containing a sequence of commands used to import, manipulate, and analyze data. In R, such a script file usually ends in the extension `.R`. At this point we will start writing and working from R scripts.

If you are using RStudio, open a new R Script using the File menu. Create a new folder/directory specifically for this Lab and save your script into the new folder. You might call it `lab0.R`.

Importantly, it is usually a good idea to tell R what directory we are working from. You do this using the `setwd()` function. If my directory for Lab 0 is `/home/peter/oeb153/lab0/`, I would put the following command in my new script (`lab0.R`) and run it in the R console.

```
> setwd("/home/peter/oeb153/lab0/")
```

Notice the directory is enclosed in double quotes. If you're using Windows, your folder path will have a hard drive and backwards slashes: `C:\Path\to\Lab0`, for example.

In most good environments for working with R, there is a keyboard shortcut for sending the current line to the console. In RStudio, you can do this by hitting `Ctrl+Return`. Try running your `setwd()` command using this shortcut.

Importing data

Real data is very rarely typed directly into R. Usually, data is imported from text files storing the data in tabular format. One common type of text file for storing data is called a comma-separated variable file (`.csv`), where each row of the table gets its own row, and each element in a row is separated by a comma.

In this Lab we'll be working with some data adapted from Gilbert JDJ, Manica A (2010) Parental care trade-offs and life history relationships in insects. American Naturalist 176: 212-226. The data are in a file called `insect_stats.csv`. Here are the first few lines of `insect_stats.csv`.

```
binomial,mass,fecundity,eggvolume
Necrobia_rufipes,8.05,89.59,0.13
Coccinella_septempunctata,26.38,700,0.6
Coleomegilla_maculata,3.44,600,0.6
Hippodamia_convergens,3.44,221,0.6
Gymnetron_antirrhini,2.89,54,0.15
```

The first line is a list of variable names or *header*. After that, each entry represents the name, mass (mg), fecundity, and egg volume (mm^3) of an insect species. These values are separated by commas.

Download `insect_stats.csv` from the course website into your Lab 0 folder. You can find the file under Labs.

To import the data into R, we can use the function `read.csv()`. Add the following command to your script and run it:

```
> insects = read.csv("insect_stats.csv", header=TRUE)
```

The `header=TRUE` option specifies that the first line of data gives the names of the variables. If you were able to run the command without throwing an error, you've successfully imported the data! Get a feel for the data with `head(insects)`.

If you get an error running the data import command, make sure that you downloaded the data file into the correct folder and that you specified the correct folder in the `setwd()` command above.

Practice questions

Try to answer the following questions about the data from Gilbert and Manica (2010). You will need to make extensive use of logical tests, a few built-in statistical functions, and vector indexing.

1. What is the mass of the most massive insect in this study?
2. Calculate the sample mean and variance of the egg volumes of insects in this study.
3. How many of the insects have a lifetime fecundity greater than 40?
4. What is the species name of the most fecund insect in this study?

5. Amongst the insects with a mass less than 10 mg, what is the average fecundity? And amongst those with mass greater than 10 mg?

You can plot the relationship between different variables using the function `plot()`. For example:

```
> plot(insects$mass, insects$eggvolume)
```

This plot would benefit from both axes being displayed on a log scale. Modify the above command to create a log-log plot of mass and egg volume. For ease of interpretation, use a base-ten logarithm. (Remember that `base` is one of the arguments to the `log()` function. See `?log`.)

There are a plethora of commands to customize plots. Check out `?plot` to see some of them.

To create a histogram, use the function `hist()` For example, to get a histogram of egg volumes:

```
> hist(insects$eggvolume,breaks=10)
```

The `breaks` option controls the number of bins in the histogram. See what information you might glean from the histogram by having fewer or more than 10 bins.