R, The Basics

R Workshop - 8/14/2018

10 Things about R to get you started:

1. History of R

R is a dialect of the *S* language that was written by John Chambers and others at Bell Labs in the 70s. In the 90s, R was developed and made available to the public with the GNU general public license. Importantly, R is *free*, meaning that you don't have to pay for it (duh), but that you have freedom to use and modify it.

2. Installing and Starting

Go to http://cran.r-project.org.

When R starts it loads some infrastructure and provides you with a prompt: `>`

This prompt is the fundamental entry point for communicating with R. We type expressions at the prompt, R evaluates these expressions, and returns output.

3. Objects in R

R is "object-oriented" meaning that we can create objects that persist within an R session and they can be manipulated.

We create objects by using the assignment operator: `<-`

What you type on the right is assigned to what you type on the left. For example:
`y <- 4` (we have assigned the value 4 to the object y)
`x <- 6` (we have assigned the value 6 to the object x)
`z <- y` (we have assigned the value of the object y to the object z, i.e. z = 4)

After assigning a value to an object, enter the name of the object to see what the value is.

Variables can start with a letter, digits, or periods. Pretty much anything. Just not a number by itself. Some examples (the second one does not work):

```
the.number.two <- 2
2 <- the.number.two
```

R is case sensitive (i.e. `A` is a different object than `a`). R is insensitive to white space though.
These two examples are treated the same in R:

```
x <- 2
x<-            2
```

To have R ignore text, use the `#` sign to make comments.
For example: `x <- 2 # this assigns the value 2 to object x.`

4. Functions in R

A major strength of R is the ability to manipulate objects using functions. A function takes an argument (aka input) and returns some value (aka output).

For example, suppose we wanted to create a list of numbers, called a *vector*. We want to create an object that is defined by the list of numbers. In R, there is a preprogrammed function `c()`, which combines or *concatenates* values to create a single object. We can create an object `x`, that is a vector of 1, 2, 3, 4, and 5 using: `x <- c(1,2,3,4,5)`.

This reads: the object x is assigned the values 1, 2, 3, 4, and 5. The function is "`c`" and the argument is `1,2,3,4,5`.

The number of values (aka elements) a vector contains is referred to as the "length". We can use the `length()` function to return this information for us. For example: `length(x)` shows that the vector `x` has 5 values or elements.

5. Referencing and Indexing Objects in R

In R, specific elements in an object are referenced by using brackets (i.e. `[` or `]`). For example:

```
x <- c(1,2,3,4,5)
x[5] # what is the fifth element in x?
x[2:4] # what are the second through fourth elements in x?
x[c(1,4)] # what are the first and fourth elements in x?
# note the difference in use between [#:#] and [c(#,#)].
```

We can also change values by indexing:

```
x[5]   <- 3 # change the fifth element in x to 5.
x[1:5] <- 0 # change the first through fifth elements in x to 0.
```

Using brackets to identify particular elements, called indexing, is VERY useful. By using indexing, we can create objects from other objects, or reference particular locations. The utility of this will be more obvious later.

6. Characters and Types of objects in R

R can also work with characters in vectors. A "character vector" is a vector of text strings:

```
c("Hugo","Desmond","Largo") # a vector with three elements.
```

We can ask R to tell us what "type" of vector a particular object is:

```
name.list <- c("Hugo","Desmond","Largo") # assign the characters to an object.
is.character(name.list) # is the object a character vector?
is.numeric(name.list) #is the object a numeric vector?
```

R has several built in character objects that can be useful:

```
LETTERS     #all upper-case letters.
letters     #all lower-case letters.
month.name  #all months by full name.
month.abb   #all months by abbreviated name.
```

Missing values are dealt with in R by NA.

```
y <- c(3,NA,10) # create a vector with a missing value.
2*y # multiple the vector by 2.
is.na(y) # which positions in y have missing values?
```

7. Matrices in R

In addition to vectors, we can create a matrix, which is a 2-dimensional representation of data. A matrix has dimensions $r\ X\ c$ which means rows by columns. The number of rows and columns a matrix has is referred to as its "order" or "dimensionality". This information is returned by using the `dim()` function. Matrices can be created by combining existing vectors using the `rbind()` and `cbind()` functions. The `rbind()` function means "row bind" and binds together vectors by rows. Think of it as stacking vectors on each other. The `cbind()` function means "column bind" and binds together vectors by rows. Think of it as placing them side by side. For both functions, however, the order of the vectors must be the same (i.e. same number of rows and columns). Let's see some examples:

```
x  <- c(6,5,4,3,2)
y  <- c(8,7,5,3,1)
m1 <- rbind(x,y) #bind x and y by row to create a 2 X 5 matrix.
m1 #just enter the name of the object to print it.
```

```
m2 <- cbind(x,y) #bind x and y by column to create a 5 X 2 matrix.
m2 #just enter the name of the object to print it.
```

We can index the matrix `m1` or `m2` by using the brackets `[ ]` with a comma between the two dimensions. Since a matrix is 2-dimensions, we can reference a specific element, an entire row, or an entire column:

```
m1[2,2] #what is the value of the element in the 2nd row, 2nd column?
m1[,2]  #what are the values in the second column?
m1[2,]  #what are the values in the second row?
```

```
m2[2,2] <- 0 #change the value to zero.
m2[2,]  <- 0 #change the second row to zeros.
m2[,2]  <- 0 #change the second column to zeros.
```

```
#note the difference between [,#] and [#,]
```

Note that `m1[2,2]`, for example, is an object, just as `m1` is an object.

Matrices can also be created from a list of numbers using the `matrix()` and `c()` functions.

```
m3 <- matrix(c(1,0,1,0,0,1,0,1,0),nrow=3,ncol=3)
m3
```

8. One of the most important functions in R: `help()`

A useful feature of R is an extensive documentation of each of the functions. To access the main R help archive online, type: `help.start()`
The `help()` function, or a simple `?`, can be used to get help about a specific function. For example: `help(c)` or `?c` returns the help page for the `c()` function.

Take a look at the help page. The first line shows you the function and the package it is written for in brackets (more on packages below). The help page provides a description, how to use it (i.e. what are the arguments), and a description of what each argument does. Further details and examples are provided as well.

Let's take a look at another function that creates sequences of numbers, the `seq( )` function. There are several ways to use the `seq( )` function. The most common are:

```
seq(from=, to=, by=) # Starts at from, ends at to, steps by by (pos or neg).
seq(from=, to=, length=) # Starts at from, ends at to, steps defined by length.
```

For example:
if we want to create an object of 5 values that starts with 1 and ends with 5, we type: `seq(from=1,to=5,by=1)`.
if we want to create an object of 5 values that starts with 1 and ends with 9, we type: `seq(from=1,to=9,by=2)`.
We could also have used the `length=` argument: `seq(from=1,to=10,length=5)`.

Since R knows that `from=` or `to=` or `by=` or `length=` are arguments, we do not have to type them in the syntax: `seq(1,9,2)` is identical to `seq(from=1,to=9,by=2)` (as far as R is concerned).

For the help function to work, you need to know the exact name of the function. If you don't know this, but have a fuzzy idea of what it might be are what you want the function to do, you can use the `help.search("fuzzy notion")` function (or just put `??` in front of the word).

For example, say you want to calculate the standard deviation for an object, but do not know the function name. Try: `help.search("standarddeviation")` or `??standarddeviation` (note the absence of a space). This returns the list of help topics that contain the phrase. We see that the standard deviation function is called `sd()`.

9. Packages and the `install.packages()` and `library()` Functions

R has MANY preprogrammed functions that are automatically loaded when you open the program. Functions are stored in "packages". Although there are many preprogrammed functions, there are even MORE functions that you can install on your own. A package in R is a collection of functions, usually written for a specific purpose.

We can see the packages available from CRAN at http://cran.r-project.org/. Just click on the "packages" link or go to https://cran.r-project.org/web/packages/index.html. As of writing this there are nearly 13,000 packages. There is a WIDE variety of packages available, this is another reason why R is awesome. If you can think it, someone has probably written a package for it in R (and if not, *you* can write one and contribute [isn't it great!]).

Take a few moments and look through the packages.

If there is a particular package you want to add, you simply use the `install.packages()` function like this: `install.packages("package name")`.

After the package is installed on your machine, you do not need to reinstall it each time you open a new session. Rather, you just need to load the package using the `library()` function like this: `library("package name")`.

For example, let's look at the sudoku package (yep, someone wrote a program to create sudoku puzzles in R). To get it we type: `install.packages("sudoku")` and then we select a "mirror". This installs the package in your local library of packages. To load the package we use the `library()` function: `library("sudoku")`. Let's take a look at what the package has: `help(package="sudoku")`.

Note that some packages require other packages for them to work. If there is an error, you need to install the additional packages.

For example, let's install the ergm package, a set of tools for estimating exponential random graph models. To get it we type: `install.packages("ergm")` and look at the additional packages that are installed. Then type: `library("ergm")`.

Some packages will not do this automatically and require that you load the library. If this is the case, R will return an error saying that a particular function is not found. For example, the function `ergmm()` in the package `latentnet` is used to fit latent space and latent space cluster random network models. Type `?ergmm` and you get an error stating that there is no documentation available. This is because the `latetnet` library has not been loaded (even if you have installed `latentnet`). Typing `install.packages("latentnet")` and `library(latentnet)` prior to `?ergmm()` will solve this problem.

Since anyone can write and contributes packages to R, it is not surprising that some packages occasionally use the same names for functions. When you have loaded libraries for packages that have conflicting functions, R will output a message indicating there is an issue. In such cases, you can unload the package using the `detach()` function. See: `?detach` for an example.

10. R Session Management

All variables created in R are stored in the "workplace". Think of it as a work bench that has a bunch of stuff on it that you have created.

To see what exists in the workplace, type: `ls()`.

We can remove specific variables with the `rm()` function. This helps clear up space (i.e. conserve memory). For example:

```
x<-seq(1,5,1) # create the object.
ls()          # see the objects.
rm(x)         # remove the object x.
ls()          # no more x.
```

To remove everything from the workplace use: `rm(list=ls())`. This is helpful for starting a session to make sure everything is cleaned out.

When you start R, it nominates one of the directories on your hard rive as a working directory, which is where it looks for user-written programs and data files.

To determine the current directory, type: `getwd()`.
You can set the working directory also by typing: `setwd("`your desired directory here`")`.

For example, if you are using Windows OS and want to set your directory to be the "C" drive, type: `setwd("C:/")`.

Or, if you are using Mac OS and want to set your directory to be a folder called "Users", `type: setwd("/Users")`.

On the Windows OS you can set R to automatically start up in your preferred working directory by right clicking on the program shortcut, choosing properties, and completing the 'Start in' field. On the Mac OS you can set the initial working directory using the Preferences menu.

To save the workspace use the `save.image()` function. This function requires a file path, a file name, and the extension ".RData" which is the format for an R workspace file.

For example, to save a workspace called "RWorkshop" to the current directory, simply type: `save.image("RWorkshop.Rdata")`. You can also write in the directory of you want to save it somewhere else. You can also do this by the pull-down menu with the File/Save option.

To load a previously saved workspace, you can either click on the file outside of R or use the `load()` function (e.g. `load("RWorkshop.Rdata")`). If you get an error, make sure you are referring the correct directory. You can also choose Load Workspace from the pull-down menu.

Note that only the objects in the workspace are saved, not the text of what you have written.

11. (Bonus!) R Studio

You may be surprised to discover how little functionality is implemented in the standard R GUI (i.e. graphical user interface). The standard R GUI implements only very rudimentary functionality through menus: reading help, managing multiple graphics windows, editing some source and data files, and some other basic functionality. There are no menu items, buttons, or palettes for loading data, transforming data, plotting data, or doing any real work with data. Commercial applications like SAS, SPSS, and Stata include user interfaces with much more functionality.

This was just the nature of working with R until some awesome human beings created RStudio. RStudio is one of several projects to build an easier-to-use GUI for R. It is a free, open-source IDE (i.e. integrated development environment) for working with R. Unlike the standard R GUI, RStudio tiles windows on the screen and puts different windows in different tabs. RStudio can be downloaded from: http://www.rstudio.com.

Go ahead and open R Studio and let's take a look.