Baby Steps with R (we all start here!)

Think of R as a graphing calculator gone wild. It can do all kinds of complex analyses in diverse fields, far more than any of us will know. But – baby steps first. This document should get you started working with R, and you can build your skills as needed later on. Some things to know:

- R stores information in *objects*.
- Objects have names, and they are case-sensitive. Names of objects may not begin with a number.
- R uses *functions* to operate on objects. A function is an object too: it's conceptually similar to the mathematical definition $y = f(x)$.
- A function takes an *object*, perhaps some additional *arguments*, and returns an answer, or maybe a plot. Most functions have very logical defaults which you don't have to specify. All function names are followed by parentheses, which may be empty, or contain arguments. In `y <- sqrt(x)`, `sqrt` is the function, and `x` is the argument that is acted upon. The answer is stored in objects `y`.
- `y <- x * 2` means take the values of `x` given, multiply them by two, and call that `y` (or more technically, *assign* the value of `y` to `x * 2`). If one value of `x` is given, then one value of `y` is returned. If a bunch of `x` values (a *vector*) are given, then `y` will be composed of a bunch of `y` values. This is one way in which R is vectorized to make computations run faster.
- <- is the *assignment operator*. It functions much like an equals sign, and you can get away with using an equals sign, but the assignment operator is better for some technical reasons.
- The *working directory* is the directory on your computer in which you are working; this is where R looks for files, and writes files. You can type `list.files()` to do a directory listing.
- The *workspace* is a bit of memory on your computer in which R stores the work you do. You can type `ls()` to see the objects in your workspace.

Getting to your Data (Working Directory)

Depending upon how you launch R, you may need to move to the directory that contains your data. The GUI provides a convenient means, or these commands work:

```
getwd()  # reports the current working directory
setwd("/Users/bryanhanson/Desktop")  # Set a new working directory on a Mac
setwd("C://Documents and Settings//Desktop")  # Set a new working directory on Windows (slashes
must be doubled!)
```

Communicating with R

>is the console prompt. Type a command after it and hit return/enter to execute the command. Recent commands are available via the up or down arrows, which contain a history of the commands. You may call one up, then use the left/right arrows to modify it and then return to carry out the modified command. The esc key will cancel whatever you are writing in the console and return you to a clean prompt. The # sign is the comment character. Everything after it is ignored. Use it to write notes to yourself in script files as a way to document your work. If you give R a command, and it returns a +, your command was not complete and it is waiting for more input, such as a final parentheses.

If you created `x`, and want to see it, just type `x` at the prompt. Watch out though, if `x` is long, it all gets dumped to the screen. Compare what happens when you do these things:

```
x <- rnorm(10)
x
rnorm(10)
```

In the last case, you just see the result; there isn't any assignment made so you can't use the result, it's merely displayed.

Creating Numbers to Play With...

When working with R, it is very often helpful to create some simple bits of data to test functions and scripts. The idea is that knowing the input data helps to understand the output and internal functioning. Here are some ways to generate numeric vectors with various characteristics:

```
x <- 1:100  # the numbers from 1 to 100
x <- c(1:100)  # another way; c = concatentate, meaning to link in a series
x <- c(0, 5, 317, 22, 99)  # any numbers you choose
x <- seq(0, 100, 5)  # means from 0 to 100 by fives, so 5, 10, 15, ... 100
x <- rnorm(100)  # a random sample of 100 from a normal distribution
x <- runif(100)  # a random sample of 100 from a uniform distribution
```

Saving Your Work (and getting it back!)

Saving graphics to a pdf file:

```
pdf("my file name.pdf", width = 10, height = 7)  # landscape orientation & size
hist(rnorm(1000))  # draws a histogram; only graphical output can go to a pdf file
dev.off()  # closes the pdf file.
```

Now you can view the pdf file. The results of `hist` did not appear in the graphics window because they were sent to the pdf file. Saving the workspace: You can save and later retrieve your entire workspace as follows. Then you can literally pick up where you left off:

```
save("my work.RData")
load("my work.Rdata")  # or you can find the file and double-click it
```

Using scripts: By far the best way to do anything beyond the most trivial task is to use a script. This allows you to write code, test it, modify it, and save it. A script file is more or less a plain text file with the extension .R so that R will recognize it. Use the GUI to create a script containing your R commands. Then you can copy and paste portions of it to the console to be executed. Or, you can execute the whole thing with the `source` command:

```
source("my script file.R")
```

You might also find it convenient to use RStudio.

Mathematical Operators

These are pretty much as expected: +, -, *, /, ^.

Simple Functions that Act on Vectors

sqrt(x) min(x) max(x) mean(x) median(x) sum(x) sd(x) summary(x)

Atomic Data Types (Modes)

- vectors – not really a type, but a term that means a 1D string of data entries. You can have a vector of any data type below.
- logical – TRUE or FALSE
- integer – you guessed it, whole numbers
- numeric – a general number; might actually be an integer too
- character – character strings, as in c("red", "blue", "green"); remember c $\neq$ character, but means to "concatenate"
- factor – a special type representing an experimental condition or level, e.g. male and female responders in a survey, treatment and control in a medical study

More Complex Data Types

- data frame – A very important concept. Conceptually a table, with observations in rows, and variables in columns. Each column may be a different atomic type or mode. Every "cell" must have a value, but that value could be NA.
- list – an arbitrary collection of almost any kind of data, no restrictions on length or type.

- matrix – A multidimensional beast, also called an array, could be any atomic type, but all entries have to be the same type (`NA` is allowed).

Special (Reserved) Words

- `NULL` – empty or undefined
- `NA` – not available; value is missing
- `NaN` – not a number, where one was expected (seen in error messages; you can't set it)
- `-Inf, +Inf` – what they look like
- `c` – concatenate, don't use as an object name!
- object names cannot begin with a number

Graphics

Graphics functions are functions that typically don't give an answer, they give the *side effect* of a plot. `R` has several different graphics systems, which vary in their abilities and advantages. The main ones are `base` graphics, `lattice` and `ggplot2`. See a separate guide about how to use `ggplot2` which in my opinion is the easiest to get started with and gives extremely good plots with well-chosen defaults.

Getting Help

Reading `R` help files take some getting used to, but the information you need is there, somewhere!

```
?sd # gives help on the function "sd"
```

If you don't know the name of the function, you have some other options, but they tend to give a lot of information. The following searches not only help files, but also mailing list archives; it does a pretty good job of ranking results:

```
RSiteSearch("my search string")
```

```
require("sos")  # installs and loads the packages 'sos'
findFn("my search string")
```

The `sos` package (last example above) is probably the best approach to finding information when you are not certain of the function names.