A first session in R

Topics covered

1. Why R?
2. starting and quitting R
3. working in a terminal
4. getting help
5. scripting

Why R?

You're here, so you see some benefit in learning R, but I'll just list a few of my reasons for using R extensively (and almost exclusively) for data analysis and graphics:

- R facilitates the realization of my work philosophy (more on that later)
- R is free and open-source. This doesn't just mean that you don't have to pay for it (which is true), but that it has a huge user base that indirectly provides support (google search), and scripts to do anything you could ever imagine wanting to do.
- Scripting is your friend.

A first session in R

Start R (Windows: Start→All Programs→Statistics Programs→R x64 3.0.1). You should see something like this (as well as some other windows):

```
R version 2.10.1 (2009-12-14)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

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.

>
```

Don't be intimidated by the blinking cursor and relatively sparse GUI–you are now using R. This begs the question, "What is R?". R is simply an engine that does things you ask it to do and then reports back to you on what it did. The > is the "command prompt" where you input commands and the blinking cursor is R telling you it is ready to do your bidding. The only complication is that R doesn't "speak" english; it has it's own language that you have to learn if you want R to understand what you are telling it. In very simple cases, this is not a problem[1]:

```
> 2+2
```

```
[1] 4
```

However, strict language guidelines can mean that commands that appear similar actually communicate very different things to R:

```
> c(1,2)
```

```
[1] 1 2
```

```
> c[1,2]
```

[1] From here, commands you should type at the command prompt are green on a gray background, and R output is black on gray. **Please type all the commands–the only way you are going to learn R is by using it, starting now.**

```
Error in c[1,2] : object of type 'special' is not subsettable
```

So our first task is to learn the language of R so that we can communicate appropriately. As we'll see later, there is a big difference between miscommunicating with R and telling R to do the wrong thing. The former is just annoying whereas the latter is the real interesting part about analyzing data.

You know how to start R, now let's quit R from the command line:

```
> q()

  Save workspace image? [y/n/c]:
```

What is this workspace image? As you work in R, the program stores a history of everything you do in the "workspace". When you quit the program, it asks if you want to save that history. If you answer y (Please don't, yet!), R will save this history into a file called .Rhistory[2] in the *current working directory* (and the program will quit).

→ For now, type c, for "**c**ancel" and you'll be back at the command prompt.

Before we save the workspace image, let's find out what the current working directory is:

```
> getwd()

  [1] ''/home/michael/Documents''
  <output will vary depending on your operating system and setup>
```

The working directory is an incredibly important concept. getwd stands for "**get w**orking **d**irectory". Loosely put, it is what R "sees" when you are working. You can imagine that R is in a room. It can see everything in that room so that you can refer directly to the things in the room: "chair", "desk", "table", "picture". It can also see things outside of the room, but only if you tell it exactly where to look. For example, "Biosciences building–front office–desk". As my current working directory is /home/michael/Documents, it's as if R is sitting in that directory and can see all the files and folders there. It can also see files and folders elsewhere as long as I tell it where to look: either relative to the current directory or as an absolute specification relative to the hard drive (for example, C:/ in Windows). I belabor a point that may be obvious to many of you because *not understanding the concept of the current working directory is one of the most common problems I see with people learning R*.

→ Now that we know where to find our .Rhistory file, quit R and save the workspace image:

```
> q()

  Save workspace image? [y/n/c]: y
```

→ Now take a look at your R history by opening the .Rhistory file (Windows: use notepad; Mac: use notepad equivalent). You'll have to go to the working directory we just found to find the file.

You'll notice that it has a list of all the commands you typed in your last session. It also saves all of the variable names and their values–basically everything from your previous session. When you start R again (from that directory), it will load the history and your new session will be a continuation of the previous one (as if you never left).

When I first started using R, I thought this was fantastic and always saved my history. Though there are limited cases where saving the history is useful (e.g., intensive calculations that you don't want to repeat), in general I think it is a BAD idea to save and reload the workspace history for a couple of reasons:

1. If you've defined and redefined objects–something you'll do a lot–you are likely to get confused as to their current state, leading to lots of cryptic errors and frustration.

[2] A leading period hides files in Windows and Linux, so you may have to show hidden files to see the R history. To view hidden files on the mac, you can download a utility or use the terminal

2. Scripts provide a clear, well-documented history of an analysis (more on this later) and prevent object confusion, thereby making the workspace history largely irrelevant.

→ So, for now, please delete the .Rhistory file, and then restart R.

→ Now let's change our working directory to a different folder. To do so, we use the command setwd():

```
> setwd
```

```
function (dir)
.Internal(setwd(dir))
<environment: namespace:base>
```

Wait, that's not what I intended to do! I forgot to give the *function* `setwd` an *argument* or parentheses. A *function* is simply a bit of `R` code that does something and an *argument* is something passed to a function so that it can do what it does. If you type any function without parentheses, `R` will show you what the code for that function is. `setwd` is a special kind of function, called an "internal" function, meaning it is relying on the guts of the program to perform its job (in this case, changing where `R` is "looking"). What about a function that isn't "internal"? The name of the function for calculating standard deviation is `sd`:

```
> sd
```

```
function (x, na.rm = FALSE)
{
    if (is.matrix(x))
        apply(x, 2, sd, na.rm = na.rm)
    else if (is.vector(x))
        sqrt(var(x, na.rm = na.rm))
    else if (is.data.frame(x))
        sapply(x, sd, na.rm = na.rm)
    else sqrt(var(as.vector(x), na.rm = na.rm))
}
<environment: namespace:stats>
```

This is one of the most powerful things about `R`: it is completely transparent! You can see exactly how any and every[3] function works. We'll come back to writing functions later, but let's take a brief look at the function `sd`. It takes two arguments, `x` and `na.rm`. You'll notice at the bottom, between the < and >, something about environment and namespace. It tells us that `sd` is part of the `stats` package (which ships with `R` as it is part of the "base" distribution). We'll talk more about packages soon.

→ Set the working directory to a useful location (perhaps a folder for this class):

```
> setwd(''/home/michael/Desktop/HandsonR13'')
```

If you type the above command in Windows, you'll get an error. This is because Windows uses the forward slash \ to delineate directories. The problem is that \ is used as an escape character in `R`. (Mac is based on Unix, so this shouldn't be a problem). To get around this on Windows systems, you have to escape the escape character so that it is recognized as a folder delineator:

```
> setwd(''C:\\home\\michael\\Documents\\Courses\\Hands on R'')
```

You'll notice that `R` doesn't output anything for either of these commands. You can double check that your working directory was changed:

```
> getwd()
```

```
[1] ''/home/michael/Documents/Courses/Hands on R''
```

It worked. You'll also notice that `getwd` is a function that doesn't take any arguments–these are common and can be quite useful so we will talk about them more soon.

[3] with the exception of "internal" functions, which are doing things that you shouldn't need to see the code for anyway

Getting help

Before we go much further, we need to learn how to get help. For example, what if we didn't know what arguments the function `setwd` expected? We can easily figure this out from within `R`, using `?` or `help()`:

```
> ?setwd
> help(setwd)
```

```
getwd                   package:base                   R Documentation

Get or Set Working Directory

Description:

     'getwd' returns an absolute filename representing the current
     working directory of the R process; 'setwd(dir)' is used to set
     the working directory to 'dir'.

Usage:

     getwd()
     setwd(dir)

Arguments:

     dir: A character string.

Value:

     'getwd' returns a character vector, or 'NULL' if the working
     directory is not available.  On Windows the path returned will use
:
```

Depending on your operating system and working environment, this will pop up a help window for the function. At first, these help windows may seem completely useless (I remember thinking so myself). But after some time, you will likely rely on them heavily, as I do now. I find the *See Also* and *Examples* sections particularly helpful. Notice that the *Usage* section tells you exactly how to execute the command (expected arguments, etc.), and the details of the required arguments are given in the *Arguments* section.

Sometimes (often) you will not know the name of the function for a particular analysis. From within `R`, `help.search()` may help you find what you are looking for. If I wanted to do a Student's t-test, but didn't know what command to use, I might type (note that the search term *must* be in quotes):

```
> help.search(''t test'')
```

The problem with this approach is that you'll often get zero or way too many hits (look at the output from the above command). The function `apropos` will return all the function names that contain the pattern in quotes:

```
> apropos(''test'')
```

```
 [1] "ansari.test"             "bartlett.test"
 [3] "binom.test"              "Box.test"
 [5] "chisq.test"              "cor.test"
 [7] "file_test"               "fisher.test"
 [9] "fligner.test"            "friedman.test"
[11] "kruskal.test"            "ks.test"
[13] "mantelhaen.test"         "mauchley.test"
[15] "mauchly.test"            "mcnemar.test"
[17] "mood.test"               "oneway.test"
[19] "pairwise.prop.test"      "pairwise.t.test"
[21] "pairwise.wilcox.test"    "poisson.test"
[23] "power.anova.test"        "power.prop.test"
[25] "power.t.test"            "PP.test"
```

```
[27] "prop.test"                "prop.trend.test"
[29] "quade.test"               "shapiro.test"
[31] "testInheritedMethods"     "testPlatformEquivalence"
[33] "testVirtual"              "t.test"
[35] ".valueClassTest"          "var.test"
[37] "wilcox.test"
```

Once you know the name of a function, you can either use the `?` to find out more, or you can use the `example` function to see the function in action (if it has an example). Try this with `t.test` and see what happens:

```
> example(t.test)
```

You can also type `help.start()` to start the interactive html version of the `R` documentation where you can browse or search for information. Try it:

```
> help.start()
```

For questions about more complex analyses, I recommend searching online instead: if you google "r cran t test", for example, you'll quickly find what you are looking for. You can also use the R site search either by visiting the link or from within `R`:

```
> RSiteSearch(``t.test'')
```

I also recommend that you download and print Jonathan Baron's R reference card–it's useful to have at your side, particularly as you begin to use `R`.

Start using R

`R` can function like a calculator. Type a few calculator computations in to see what happens. For example, to calculate the answer to life the universe and everything:

```
> 12*14/(2^2)
```

```
 [1] 42
```

`R` understands standard mathematical "operators", like multiplication, division, addition, subtraction, etc. Note the output, which begins with `[1]`, indicating that `42` is the *first* thing returned by `R` for your command.

Let's move beyond the calculator and do useful things like make "variables", that is, objects with names that hold things:

```
> a <- 2
```

Here we've created a variable `a` with the value 2. We did so by using the "assignment operator", `<-`. It looks a bit like an arrow pointing from `2` to `a`. You can read the command above, "a gets 2", or "a is assigned the value 2". If we look at a, we see its value is now 2:

```
> a
```

```
 [1] 2
```

Now we can do fun things with `a`:

```
> a*a; a*2; a^2
```

```
 [1] 4
 [1] 4
 [1] 4
```

The semicolon allows you to put more than one command on a single line, but each command is executed separately and generates its own output (which happens to be `4` in every case).

So far, we've dealt with single numbers, or "scalars". These are a special type of "vector" (namely, a vector with only one element). The most common way to create vectors, which are lists of items, is with the "**c**oncatenate" function:

```
> b <- c(1,2)
> b
```

```
[1] 1 2
```

We've assigned to the variable `b` the stuff to the right of the `<-`. `c(1,2)` simply says, "make a vector with `1` followed by `2`.

In-class exercises

Now it's time to get your feet wet.

→ Start with the calculator chapter from Owen's R Guide.

Run through all the commands and think about how R is executing them – this is your chance to play! Once you have gone through them yourself, download my script which goes through them step by step and explains some of the concepts in more detail. [When you go to the above link, it will display the file in the browser. I recommend Saving as Owen-calculator.R in your play directory]

Homework

1. Print out Baron's R-reference card and bring to class with you every week, starting next week.
2. Write a well-documented script to do the Exercises at the end of the Owen calculator chapter. Try to adhere to the scripting guidelines on the next page and feel free to use my script for the Owen calculator chapter as a starting point.

Some general scripting guidelines

For specific coding style recommendations agreed on in general by the R community, see Google's R style guide. Below are my thoughts and recommendations on this.

Use meaningful identifiers. When you are first developing code, you'll find yourself using generic identifiers for vectors, data frames, variables, etc. (e.g., "data", "junk", "a"). While expedient in the short-term, generic names like these will only cause lost time and oodles of frustration in the long term.

Comment profusely. Commenting code is essential. I am continually surprised by how quickly I forget what I've done and how I did it. Even though you have a better memory than I do, comment your code for the benefit of everyone you'll share it with, even if you don't envision sharing it when you write it. One of the great strengths of R (or any scripted language) is that code can be shared!

Use a consistent style. Whether it be where you place comments, how you identify sections of your script, or how you space text, be consistent. The goal here is code that is legible to someone else and to yourself 5-10 years from now.

Recommended script layout. I follow a consistent style when starting new scripts so that I can keep track of what they are, what they do, what they depend on, and their modification history.

```
# filename.R
#
#  This script does some neat things by relying on interesting techniques ...
#
#  Depends: data.txt
#           functions.R
#           other_analysis.R
#
#  created 6-10-10 by M. Dillon
#
#  last modified:
#             by:
#
#  More detailed description if necessary--requirements, structure of data,
#    references, etc.  Much of this may go in comments throughout the script.
##############################################################################

## libraries
library(Hmisc)

## non-standard functions
source(''functions.R'')

the rest of the script
```