Using R

September 7, 2013

This will cover some of the more advanced features of R that were not covered, or were treated lightly in the R bootcamp. It's a bit of a mish-mash because I'm assuming you have a working familiarity with the first 8 or so units from the bootcamp.

References:

- Adler
- Chambers
- R intro manual on CRAN (R-intro).
- Venables and Ripley, Modern Applied Statistics with S
- Murrell, Introduction to Data Technologies.

I'm going to try to refer to R syntax as statements, where a statement is any code that is a valid, complete R expression. I'll try not to use the term *expression*, as this actually means a specific type of object within the R language.

One of my goals in our coverage of R is for us to think about why things are the way they are in R. I.e., what principles were used in creating the language.

1 Interacting with the operating system from R

I'll assume everyone knows about the following functions/functionality in R:
getwd(), setwd(), source(), pdf(), save(), save.image(), load()

- To run UNIX commands from within R, use *system()*, as follows, noting that we can save the result of a system call to an R object:

```
system("ls -al")  # knitr/Sweave doesn't seem to show the output of sys

files <- system("ls", intern = TRUE)

files[1:5]

## [1] "best-practices.png" "branchcommit.png"   "cache"
## [4] "commit_anatomy.png" "example.bashrc"
```

- There are also a bunch of functions that will do specific queries of the filesystem, including

```
file.exists("file.txt")

list.files("~/research")
```

- To get some info on the system you're running on:

```
Sys.info()

##                                          sysname
##                                          "Linux"
##                                          release
##                              "3.2.0-43-generic"
##                                          version
## "#68-Ubuntu SMP Wed May 15 03:33:33 UTC 2013"
##                                         nodename
##                                       "smeagol"
##                                          machine
##                                         "x86_64"
##                                            login
##                                       "unknown"
##                                             user
##                                      "paciorek"
##                                              effective_user
##                                                  "paciorek"
```

- To see some of the options that control how R behaves, try the *options()* function. The *width* option changes the number of characters of width printed to the screen, while the *max.print* option prevents too much of a large object from being printed to the screen. The *digits* option changes the number of digits of numbers printed to the screen (but be careful as this can be deceptive if you then try to compare two numbers based on what you see on the screen).

```
# options() # this would print out a long list of options

options()[1:5]

## $add.smooth
## [1] TRUE
##
## $bitmapType
## [1] "cairo"
##
## $browser
## [1] "xdg-open"
##
## $browserNLdisabled
## [1] FALSE
##
## $CBoundsCheck
## [1] FALSE


options()[c("width", "digits")]

## $width
## [1] 75
##
## $digits
## [1] 4
```

```
# options(width = 120) # often nice to have more characters on screen

options(width = 55)  # for purpose of making the pdf of this document

options(max.print = 5000)

options(digits = 3)

a <- 0.123456
b <- 0.1234561

a

## [1] 0.123

b

## [1] 0.123

a == b

## [1] FALSE
```

- Use `Ctrl-C` to interrupt execution. This will generally back out gracefully, returning you to a state as if the command had not been started. Note that if R is exceeding memory availability, there can be a long delay. This can be frustrating, particularly since a primary reason you would want to interrupt is when R runs out of memory.
- The R mailing list archives are very helpful for getting help - always search the archive before posting a question. More info on where to find R help in Unit 5 on debugging.
  - *sessionInfo()* gives information on the current R session - it's a good idea to include this information (and information on the operating system such as from *Sys.info()*) when you ask for help on a mailing list

```
sessionInfo()

## R version 3.0.1 (2013-05-16)
## Platform: x86_64-pc-linux-gnu (64-bit)
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8
##  [2] LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8
##  [4] LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8
##  [6] LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=C
##  [8] LC_NAME=C
##  [9] LC_ADDRESS=C
## [10] LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8
## [12] LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
## [6] base
##
## other attached packages:
## [1] knitr_1.2 SCF_1.1-1
##
## loaded via a namespace (and not attached):
## [1] digest_0.6.3   evaluate_0.4.3 formatR_0.7
## [4] stringr_0.6.2  tools_3.0.1
```

- Any code that you wanted executed automatically when starting R can be placed in *~/.Rprofile* (or in individual *.Rprofile* files in specific directories). This could include loading packages (see below), sourcing files with user-defined functions (you can also put the function code itself in *.Rprofile*), assigning variables, and specifying options via *options()*.

- You can have an R script act as a shell command (like running a bash shell script) as follows.

1. Write your R code in a text file, say *file.R*.
2. As the first line of the file, include `#!/usr/bin/Rscript` (like `#!/bin/bash` in a bash shell file, as seen in Unit 2).
3. Make the R code file executable with *chmod*.
4. Run the script from the command line: `./file.R`

If you want to pass arguments into your script, you can do so as long as you set up the R code to interpret the incoming arguments:

```r
args <- commandArgs(TRUE)
# now args is a character vector containing the arguments.
# Suppose the first argument should be interpreted as a number

# and the second as a character string and the third as a boolean:
numericArg <- as.numeric(args[1])
charArg <- args[2]
logicalArg <- as.logical(args[3]
```

2 Packages

One of the killer apps of R is the extensive collection of add-on packages on CRAN (www.cran.r-project.org) that provide much of R's functionality. To make use of a package it needs to be installed on your system (using *install.packages()* once only) and loaded into R (using *library()* every time you start R).

Some packages are *installed* by default with R and of these, some are *loaded* by default, while others require a call to *library()*. For packages I use a lot, I install them once and then load them automatically every time I start R using my *~/.Rprofile* file.

Loading packages You can use *library()* to either (1) make a package available (loading it) or (2) to see all the installed packages.

```r
library(fields)

## Loading required package:  methods
## Loading required package:  spam
## Spam version 0.29-3 (2013-04-23) is loaded.
## Type 'help( Spam)' or 'demo( spam)' for a short introduction
## and overview of this package.
## Help for individual functions is also obtained by adding the
## suffix '.spam' to the function name, e.g.  'help( chol.spam)'.
##
## Attaching package:  'spam'
## The following object is masked from 'package:base':
##
##    backsolve, forwardsolve
## Loading required package:  maps

library(help = fields)
# library() # I don't want to run this on SCF because
# so many are installed
```

Notice that some of the packages are in a system directory and some are in my home directory. Packages often depend on other packages. In general, if one package depends on another, R will load the dependency, but if the dependency is installed locally (see below), R may not find it automatically and you may have to use *library()* to load the dependency first. *.libPaths()* shows where R looks for packages on your system and searchpaths() shows where individual packages are loaded from.

```
.libPaths()

## [1] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0"
## [2] "/server/linux/lib/R/3.0/x86_64/site-library"
## [3] "/usr/lib/R/site-library"
## [4] "/usr/lib/R/library"

searchpaths()

##  [1] ".GlobalEnv"
##  [2] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0/field
##  [3] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0/maps"
##  [4] "/accounts/gen/vis/paciorek/R/x86_64-pc-linux-gnu-library/3.0/spam"
##  [5] "/usr/lib/R/library/methods"
##  [6] "/server/linux/lib/R/3.0/x86_64/site-library/knitr"
##  [7] "/usr/lib/R/library/stats"
##  [8] "/usr/lib/R/library/graphics"
##  [9] "/usr/lib/R/library/grDevices"
## [10] "/usr/lib/R/library/utils"
## [11] "/usr/lib/R/library/datasets"
## [12] "/server/linux/lib/R/3.0/x86_64/site-library/SCF"
## [13] "Autoloads"
## [14] "/usr/lib/R/library/base"
```

Installing packages If a package is on CRAN but not on your system, you can install it easily (usually). You don't need root permission on a machine to install a package (though sometimes you run into hassles if you are installing it just as a user, so if you have administrative privileges it may help to use them).

```
install.packages("fields", lib = "~/Rlibs")  # ~/Rlibs needs to exist!
```

You can also download the zipped source file from CRAN and install from the file; see the help page for *install.packages()*.

Accessing objects from packages The objects in a package (primarily functions, but also data) are in their own workspaces, and are accessible after you load the package using *library()*, but are not directly visible when you use *ls()*. We'll talk more about this when we talk about scope and environments. If we want to see the objects in one of the other workspaces, we can do the following:

```
search()

##  [1] ".GlobalEnv"         "package:fields"
##  [3] "package:maps"       "package:spam"
##  [5] "package:methods"    "package:knitr"
##  [7] "package:stats"      "package:graphics"
##  [9] "package:grDevices"  "package:utils"
## [11] "package:datasets"   "package:SCF"
## [13] "Autoloads"          "package:base"

# ls(pos = 7)  # for the stats package
ls(pos = 7)[1:5]  # just show the first few
```

```
## [1] "acf"          "acf2AR"      "add1"         "addmargins"
## [5] "add.scope"

ls("package:stats")[1:5]  # equivalent

## [1] "acf"          "acf2AR"      "add1"         "addmargins"
## [5] "add.scope"
```

3 Objects in R

3.1 Assignment and coercion

We assign into an object using either '=' or '<-'. A rule of thumb is that for basic assignments where you have an object name, then the assignment operator, and then some code, '=' is fine, but otherwise use '<-'.

```
out = mean(rnorm(7))  # OK
system.time(out = rnorm(10000))

## Error: unused argument (out = rnorm(10000))

# NOT OK, system.time() expects its argument to be a
# complete R expression
system.time(out <- rnorm(10000))

##    user  system elapsed
##   0.000   0.000   0.001
```

Let's look at these examples to understand the distinction between '=' and '<-' when passing arguments to a function.

```
mean

## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x4263ef0>
## <environment: namespace:base>
```

```r
x <- 0
y <- 0
out <- mean(x = c(3, 7))  # the usual way to pass an argument to a function
out <- mean(x <- c(3, 7))  # this is allowable, though perhaps not useful; 
out <- mean(y = c(3, 7))

## Error:  argument "x" is missing, with no default

out <- mean(y <- c(3, 7))
```

What can you tell me about what is going on in each case above?

One situation in which you want to use '<-' is if it is being used as part of an argument to a function, so that R realizes you're not indicating one of the function arguments, e.g.:

```r
mat <- matrix(c(1, NA, 2, 3), nrow = 2, ncol = 2)
apply(mat, 1, sum.isna <- function(vec) {
    return(sum(is.na(vec)))
})

## [1] 0 1

# What is the side effect of what I have done here?
apply(mat, 1, sum.isna = function(vec) {
    return(sum(is.na(vec)))
})  # NOPE

## Error:  argument "FUN" is missing, with no default
```

R often treats integers as numerics, but we can force R to store values as integers:

```r
vals <- c(1, 2, 3)
class(vals)

## [1] "numeric"

vals <- 1:3
class(vals)

## [1] "integer"
```

```
vals <- c(1L, 2L, 3L)
vals

## [1] 1 2 3

class(vals)

## [1] "integer"
```

We convert between classes using variants on *as()*: e.g.,

```
as.character(c(1, 2, 3))

## [1] "1" "2" "3"

as.numeric(c("1", "2.73"))

## [1] 1.00 2.73

as.factor(c("a", "b", "c"))

## [1] a b c
## Levels: a b c
```

Some common conversions are converting numbers that are being interpreted as characters into actual numbers, converting between factors and characters, and converting between logical TRUE/FALSE vectors and numeric 1/0 vectors. In some cases R will automatically do conversions behind the scenes in a smart way (or occasionally not so smart way). We'll see implicit conversion (also called coercion) when we read in characters into R using *read.table()* - strings are often automatically coerced to factors. Consider these examples of implicit coercion:

```
x <- rnorm(5)
x[3] <- "hat"  # What do you think is going to happen?
indices = c(1, 2.73)
myVec = 1:10
myVec[indices]

## [1] 1 2
```

In other languages, converting between different classes is sometimes called *casting* a variable.

Here's an example we can work through that will help illustrate how type conversions occur behind the scenes in R.

```
n <- 5
df <- data.frame(rep("a", n), rnorm(n), rnorm(n))
apply(df, 1, function(x) x[2] + x[3])

## Error:  non-numeric argument to binary operator

# why does that not work?
apply(df[, 2:3], 1, function(x) x[1] + x[2])

## [1] -0.655 -0.360 -2.855  0.433  1.978

df2 <- data.frame(cbind(rep("a", n), rnorm(n), rnorm(5)))
apply(df2[, 2:3], 1, function(x) x[1] + x[2])

## Error:  non-numeric argument to binary operator
```

3.2 Type vs. class

You should be familiar with vectors as the basic data structure in R, with character, integer, numeric, etc. classes. Objects also have a type, which relates to what kind of values are in the objects and how objects are stored internally in R (i.e., in C).

Let's look at Adler's Table 7.1 to see some other types.

```
a <- data.frame(x = 1:2)
class(a)

## [1] "data.frame"

typeof(a)

## [1] "list"

m <- matrix(1:4, nrow = 2)
class(m)

## [1] "matrix"

typeof(m)

## [1] "integer"
```

Everything in R is an object and all objects have a class. For simple objects class and type are often closely related, but this is not the case for more complicated objects. The class describes what the object contains and standard functions associated with it. In general, you mainly need to know what class an object is rather than its type. Classes can *inherit* from other classes; for example, the *glm* class inherits characteristics from the *lm* class. We'll see more on the details of object-oriented programming in the R programming unit.

We can create objects with our own defined class.

```
me <- list(firstname = "Chris", surname = "Paciorek")
class(me) <- "personClass"  # it turns out R already has a 'person' class
class(me)

## [1] "personClass"

is.list(me)

## [1] TRUE

typeof(me)

## [1] "list"

typeof(me$firstname)

## [1] "character"
```

3.3 Information about objects

Some functions that give information about objects are:

```
is(me, "personClass")

## [1] TRUE

str(me)

## List of 2
##  $ firstname: chr "Chris"
##  $ surname  : chr "Paciorek"
##  - attr(*, "class")= chr "personClass"
```

```r
attributes(me)

## $names
## [1] "firstname" "surname"
##
## $class
## [1] "personClass"

mat <- matrix(1:4, 2)
class(mat)

## [1] "matrix"

typeof(mat)

## [1] "integer"

length(mat)

## [1] 4

# recall that a matrix can be thought of as a vector
# with dimensions
attributes(mat)

## $dim
## [1] 2 2

dim(mat)

## [1] 2 2
```

Attributes are information about an object attached to an object as something that looks like a named list. Attributes are often copied when operating on an object. This can lead to some weird-looking formatting:

```r
x <- rnorm(10 * 365)
qs <- quantile(x, c(0.025, 0.975))
qs
```

```
##   2.5% 97.5%
## -1.94  1.97

qs[1] + 3

## 2.5%
## 1.06
```

Thus in an subsequent operations with *qs*, the *names* attribute will often get carried along. We can get rid of it:

```
names(qs) <- NULL
qs

## [1] -1.94  1.97
```

A common use of attributes is that rows and columns may be named in matrices and data frames, and elements in vectors:

```
row.names(mtcars)[1:6]

## [1] "Mazda RX4"           "Mazda RX4 Wag"
## [3] "Datsun 710"          "Hornet 4 Drive"
## [5] "Hornet Sportabout" "Valiant"

names(mtcars)

##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec"
##  [8] "vs"   "am"   "gear" "carb"

mat <- data.frame(x = 1:2, y = 3:4)
row.names(mat) <- c("first", "second")
mat

##        x y
## first  1 3
## second 2 4

vec <- c(first = 7, second = 1, third = 5)
vec["first"]

## first
##     7
```

3.4 The workspace

Objects exist in a workspace, which in R is called an environment.

```
# objects() # what objects are in my workspace
identical(ls(), objects())  # synonymous

## [1] TRUE

dat <- 7
dat2 <- 9
subdat <- 3
obj <- 5
obj2 <- 7
objects(pattern = "^dat")

## [1] "dat"  "dat2"

rm(dat2, subdat)
rm(list = c("obj", "obj2"))
# a bit confusing - the 'list' argument should be a
# character vector
rm(list = ls(pattern = "^dat"))
exists("dat")  # can be helpful when programming

## [1] FALSE

dat <- rnorm(5e+05)
object.size(dat)

## 4000040 bytes

print(object.size(dat), units = "Mb")  # this seems pretty clunky!

## 3.8 Mb

# but we'll understand why when we see S3 classes in
# detail
rm(list = ls())  # what does this do?
```

3.5 Some other details

Special objects There are also some special objects, which often begin with a period, like hidden files in UNIX. One is *.Last.value*, which stores the last result.

```
rnorm(10)

##  [1]  1.372  2.569 -1.556 -0.228 -1.867  0.149 -0.264
##  [8]  0.807 -0.818 -0.509

# .Last.value # this should return the 10 random
# normals but knitr is messing things up, commented
# out here
```

Scientific notation R uses the syntax *"xep"* to mean $x * 10^p$.

```
x <- 1e+05
log10(x)
y <- 1e+05
x <- 1e-08
```

Information about functions To get help on functions (I'm having trouble evaluating these with knitr, so just putting these in as text here):

```
?lm # or help(lm)
help.search('lm')
apropos('lm')
help('[[') # operators are functions too
args(lm)
```

Strings and quotation Working with strings and quotes (see `?Quotes` in R). Generally one uses double quotes to denote text. If we want a quotation symbol in text, we can do something like the following, either combining single and double quotes or escaping the quotation:

```
ch1 <- "Chris's\n"
ch2 <- 'He said, "hello."\n'
ch3 <- "He said, \"hello.\"\n"
```

Be careful when cutting and pasting from documents that are not text files as you may paste in something that looks like a single or double quote, but which R cannot interpret as a quote because it's some other ASCII quote character.

4 Working with data structures

4.1 Lists and dataframes

Extraction You extract from lists with "*[[*" or with "*[*"

```
x <- list(a = 1:2, b = 3:4, sam = rnorm(4))
x[[2]]  # extracts the indicated component, which can be anything, in this 

## [1] 3 4

x[c(1, 3)]  # extracts subvectors, which since it is a list, will also be a

## $a
## [1] 1 2
##
## $sam
## [1]  0.00966  0.35847  1.83673 -1.11176
```

When working with lists, it's handy to be able to use the same function on each element of the list:

```
lapply(x, length)

## $a
## [1] 2
##
## $b
## [1] 2
##
## $sam
## [1] 4

sapply(x, length)  # returns things in a user-friendly way
```