Day One: R Basics

Dillon Niederhut

July 27, 2015

Pre-introduction

You should start by having your class go to our github page at github.com/dlab-berkeley/R-for-Data-Science to get the course materials either via:

1. `git clone https://github.com/dlab-berkeley/R-for-Data-Science.git`; or
2. clicking the 'download zip' button on the right hand side of the screen

The students won't need these materials today, but they will for the rest of the workshop. While everything is downloading, you can go on to:

Introduction to the class

> side note - these materials are meant to be guides for you. Your students will retain more of this content if they type these commands themselves than if they read them off of the slidedeck. That being said, at any time, you can create a slide deck by changing `output:` to be `html_slides` instead of `pdf_document`.

It is a good idea to start off the class by asking folks why they want to learn R. Common responses include:

1. Stata/SPSS/Matlab is too expensive
2. I saw a pretty graph someone made in R
3. My field uses analytical packages written for R
4. I have a deep and burning desire for open and reproducible research

The outline below is designed to give each of these kinds of students the tools they need to get what they want out of R while avoiding common pitfalls. As the instructor, you should draw on your own experience to include further examples and advice, especially for students who do not fall into one of the four categories above.

Introduction to R

It may also be helpful to start off with a little bit of background knowledge about R. I find that explicitly informing students about the design principles of a language is a quick way to bootstrap their intuitions about how to use that language. R, for example, is a very old language whose objective was to allow scientists to quickly and interactively conduct statistical tests when the only other options at the time were:

1. Compile a whole program in C or FORTRAN; or,
2. Do the math yourself with a pencil and a piece of paper

Obviously, neither of these is optimal, but what might not be obvious is that they both share the same problems; they require lots of human time, and those humans have to be very knowledgeable about the mathematical principles underlying statistical computation (e.g. that even simple functions have multiple implementations to balance accuracy/efficiency for different input values).

The good news is that very complicated processes like logistic regression are a single command in R. The bad news is that R is typically not concerned with being logical or conistent. If you find yourself wanting to tear your hair out, this is **normal**.

Object Oriented Programming

In the grand scheme of computer software, object orientation is a way of organizing code such that it is easy to update without breaking. This means grouping functions that serve a similar purpose into hierarchies. However, stating it this way is confusing and abstract.

You can think about it this way: a soccer ball is an object. So is a basketball. They share a lot of things in common. It's simpler to know that balls generaly bounce than to explicitly declare for every ball I ever see in my entire life whether it bounces or not. I can't bounce you, for example, but you didn't need to tell me that when I met you. If I came to believe that people were bounce-able, I would update my idea of people generally, not every person specifically.

We call things like you and basketball `objects`, and they are in `classes` like human and ball. If I want to create a new object, like a football, I don't have to declare every single thing there is to know about footballs. I can say it `inherits attributes` from the `class` ball, except that it's an oblate spheroid instead of a sphere. Easy.

> side note - if you are coming from C++ or Java, be warned that objects in R do not have methods that are accessible with dot notation (in fact, the `.` is used just like `_`)

everything in R is an object

yes, even the commands, just watch

```
ls
```

```
## function (name, pos = -1L, envir = as.environment(pos), all.names = FALSE,
##     pattern, sorted = TRUE)
## {
##     if (!missing(name)) {
##         pos <- tryCatch(name, error = function(e) e)
##         if (inherits(pos, "error")) {
##             name <- substitute(name)
##             if (!is.character(name))
##                 name <- deparse(name)
##             warning(gettextf("%s converted to character string",
##                 sQuote(name)), domain = NA)
##             pos <- name
##         }
##     }
##     all.names <- .Internal(ls(envir, all.names, sorted))
##     if (!missing(pattern)) {
##         if ((ll <- length(grep("[", pattern, fixed = TRUE))) &&
##             ll != length(grep("]", pattern, fixed = TRUE))) {
##             if (pattern == "[") {
##                 pattern <- "\\["
##                 warning("replaced regular expression pattern '[' by  '\\\\['")
##             }
##             else if (length(grep("[^\\\\]\\[<-", pattern))) {
##                 pattern <- sub("\\[<-", "\\\\\\[<-", pattern)
##                 warning("replaced '[<-' by '\\\\[<-' in regular expression pattern")
##             }
##         }
```

```
##         grep(pattern, all.names, value = TRUE)
##     }
##     else all.names
## }
## <bytecode: 0x7fe7351e23b8>
## <environment: namespace:base>
```

`ls`, like basketball, is a specific thing with a `name` and stuff inside it that makes it `ls` and not dillon niederhut. in this particular instance, we are looking at the function that tells you what `objects` are in your `environment`

until we get to functional programming, your `environment` is just R plus whatever you put in R

in R, you store objects with names with the <- operator

just like you need names to tell things apart, R does too

```
my.name <- dir
my.name
```

```
## function (path = ".", pattern = NULL, all.files = FALSE, full.names = FALSE,
##     recursive = FALSE, ignore.case = FALSE, include.dirs = FALSE,
##     no.. = FALSE)
## .Internal(list.files(path, pattern, all.files, full.names, recursive,
##     ignore.case, include.dirs, no..))
## <bytecode: 0x7fe736781558>
## <environment: namespace:base>
```

names must be unique

everytime you give an `object` a `name`, it removes anything that already had that `name` from your environment

```
my.name <- dir()
my.name
```

```
## [1] "data"                     "examples"
## [3] "feedback_cleaner.R"       "instructor"
## [5] "LICENSE"                  "R-for-Data-Science.Rproj"
## [7] "README.md"
```

you see those parentheses? that means you are calling an object (here, it's a function evaluator) on `dir`.

classes in R

because it is code to be evalueated, `dir` belongs in a class called 'functions'

```
class(dir)
```

```
## [1] "function"
```

functions all have the same basic structure

`function(arguments)`, where the arguments are other objects, like

```
sum(1,2,3)
```

```
## [1] 6
```

`1,2,3` are also objects, with a class of their own

when you call a function, it looks at the classes of the things you are calling it on to figure out how to behave

in much the same way, if my function is to move things from point A to point B, the way I might do that to a basketball is different from the way I might do that to you

what kind of class do you think `1` is?

more bad news

R started out as a functional programming language (more on this later), to which object orientation was later added

this means that R doesn't know that some things are objects, because they predate the addition of class systems

```
is.object(sum)
```

```
## [1] FALSE
```

most of R uses what are called S3 methods, which have no rules except be easy to use. this can make them wildly inconsistent, even to the point where a single function will have multiple sets of rules for how it can be called (you'll see this in day 3).

> as a side note, there is also no agreement about how to name things, so you'll likely see a mixture of snake_case and CamelCase, based on the preferences of the person who originally wrote the function

living in R

figure out where you are with

```
getwd()
```

```
## [1] "/Users/dillon/Dropbox/dlab/workshops/R-for-Data-Science"
```

like in Unix, in R you are always in a directory

your actions are all relative to that directory

tell R where you would like it to be with

```r
setwd("/Users/dillonniederhut/Dropbox/dlab/R-for-Data-Science")
```

find out what's in your directory with

```r
dir()
```

```
## [1] "data"                    "examples"
## [3] "feedback_cleaner.R"      "instructor"
## [5] "LICENSE"                 "R-for-Data-Science.Rproj"
## [7] "README.md"
```

find out what's in your environment with

in R, you are always in an environment (more on scoping in day 4)

```r
ls()
```

```
## [1] "my.name"
```

our environment is currently empty

```r
test <- "I have no idea what I'm doing"
ls()
```

```
## [1] "my.name" "test"
```

we can clean our environment with

```r
rm(list = ls())
exists(test)
```

you can pull documentation with ?

```r
?exists
```

and search the help pages with ??

```r
??exists
```

you can get a quick example with

```r
example(exists)
```

```
##
## exists> ##  Define a substitute function if necessary:
## exists> if(!exists("some.fun", mode = "function"))
## exists+   some.fun <- function(x) { cat("some.fun(x)\n"); x }
##
## exists> search()
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
##
## exists> exists("ls", 2) # true even though ls is in pos = 3
## [1] TRUE
##
## exists> exists("ls", 2, inherits = FALSE) # false
## [1] FALSE
##
## exists> ## These are true (in most circumstances):
## exists> identical(ls,   get0("ls"))
## [1] TRUE
##
## exists> identical(NULL, get0(".foo.bar.")) # default ifnotfound = NULL (!)
## [1] TRUE
##
## exists> ## Don't show:
## exists> stopifnot(identical(ls, get0("ls")),
## exists+           is.null(get0(".foo.bar.")))
##
## exists> ## End(Don't show)
## exists>
## exists>
## exists>
```

when you kind of remember what you are looking for, try

```r
apropos('lm')
```

```
##  [1] ".__C__anova.glm"      ".__C__anova.glm.null" ".__C__glm"
##  [4] ".__C__glm.null"       ".__C__lm"             ".__C__mlm"
##  [7] ".__C__optionalMethod" ".colMeans"            ".lm.fit"
## [10] "colMeans"             "confint.lm"           "contr.helmert"
## [13] "dummy.coef.lm"        "getAllMethods"        "glm"
## [16] "glm.control"          "glm.fit"              "KalmanForecast"
## [19] "KalmanLike"           "KalmanRun"            "KalmanSmooth"
## [22] "kappa.lm"             "lm"                   "lm.fit"
## [25] "lm.influence"         "lm.wfit"              "model.matrix.lm"
## [28] "nlm"                  "nlminb"               "predict.glm"
## [31] "predict.lm"           "residuals.glm"        "residuals.lm"
## [34] "summary.glm"          "summary.lm"
```

The power of R is its extensibility

many people write clever software that makes R smarter/better/faster/stronger

you can install these packages with

```
install.packages("Amelia")
```

and include them in your environment with

```
library(Amelia)
```

```
## Loading required package: Rcpp
## ##
## ## Amelia II: Multiple Imputation
## ## (Version 1.7.3, built: 2014-11-14)
## ## Copyright (C) 2005-2016 James Honaker, Gary King and Matthew Blackwell
## ## Refer to http://gking.harvard.edu/amelia/ for more information
## ##
```

note that when you are installing something, you give R a bunch of letters to search CRAN for, which is why it's in quotes

but when you pull it into your environment, you are calling a function on a `name`, which is why it isn't in quotes

if you try to call `library` on package that you haven't downloaded, R will fuss at you

```
library(supercalifragilisticexpialedocious)
```

Math in R

R can be a calculator

```
2 + 2
```

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

```
2 - 2
```

```
## [1] 0
```

```r
2 * 2
```

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

```r
2 %% 2
```

```
## [1] 0
```

```r
2 %/% 2
```

```
## [1] 1
```

```r
2 / 2
```

```
## [1] 1
```

```r
2 ** 2
```

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

```r
2 ** .5
```

```
## [1] 1.414214
```

```r
2 ** -1
```

```
## [1] 0.5
```

R does a few more complicated things

```r
abs(-2)
```

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

```r
pi
```

```
## [1] 3.141593
```

```r
round(pi,digits = 2)
```

```
## [1] 3.14
```

```r
sign(-2)
```

```
## [1] -1
```

```r
log(2)
```

```
## [1] 0.6931472
```

```r
log10(2)
```

```
## [1] 0.30103
```

```r
cos(pi)
```

```
## [1] -1
```

R also handles logic tables and testing

```r
TRUE & TRUE
```

```
## [1] TRUE
```

```r
TRUE | FALSE
```

```
## [1] TRUE
```

```r
xor(TRUE,FALSE)
```

```
## [1] TRUE
```

```r
! FALSE
```

```
## [1] TRUE
```

```r
1 & 1
```

```
## [1] TRUE
```

```r
1 & 0
```

```
## [1] FALSE
```

```r
!0
```

```
## [1] TRUE
```

Data Types

R differentiates between different types of data

for example, the boolean and numeric values above

```r
class(TRUE)
```

```
## [1] "logical"
```

```r
class(1)
```

```
## [1] "numeric"
```

you could also use `mode` to get the type of an object

this will mean later, when you try to call `mode` to get the most frequently occurring level of a variable, you will be frustrated and sad

don't dislike the messenger

you will likely only ever deal with five flavors of data in R, which are stored as

three data types

```r
class(FALSE)
```

```
## [1] "logical"
```

```r
class(pi)
```

```
## [1] "numeric"
```

```r
class("Look mama I'm letters")
```

```
## [1] "character"
```

```r
class(as.Date("2015-07-27"))
```

```
## [1] "Date"
```

```r
class(factor(c('undergraduate','graduate','professor','staff')))
```

```
## [1] "factor"
```

> side note - by default, R stores everything as doubles (64 bit floating point numbers) which makes R very memory hungry. You can force it use an integer type with the `L` operator, like: `class(1L)` == integer

we've already dealt a lot with numerics above, so let's talk about

Boolean data

logical values pretty much act like numerics

```
TRUE + TRUE
```

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

```
2 & 1
```

```
## [1] TRUE
```

```
TRUE * TRUE
```

```
## [1] 1
```

```
2 & -1
```

```
## [1] TRUE
```

this can make it easy to use if/then statements, as `if x` evaluates to `TRUE` if it is anything other than zero

```
if (9001) print('This is evaluated as a boolean value')
```

```
## [1] "This is evaluated as a boolean value"
```

also, any vector (we'll talk about these below) multiplied by a boolean vector has all of its false values set to zero, which can be helpful for summing and average only specific cases

Character data

character handling in R is fairly close to character handling in a Unix terminal

```
my.character <- paste("Hey", "momma", "I'm", "a", "string")
my.character
```

```
## [1] "Hey momma I'm a string"
```

whitespace is the default separater in the paste function, if you don't want this, use `paste0()`

```
substr(my.character,1,4)
```

```
## [1] "Hey "
```

note here that R is not a zero-indexed language

```r
substr(my.character,1,4) <- "Yes "
my.character
```

```
## [1] "Yes momma I'm a string"
```

you can separate characters with

```r
strsplit(my.character, ' ')
```

```
## [[1]]
## [1] "Yes"    "momma"  "I'm"    "a"      "string"
```

you can substitute with

```r
gsub('.', 'X', my.character)
```

```
## [1] "XXXXXXXXXXXXXXXXXXXXXX"
```

R here calls Perl's regex library, where `.` is a special shorthand for "anything"

to be safe, put it in brackets

```r
gsub('[.]', 'X', my.character)
```

```
## [1] "Yes momma I'm a string"
```

```r
gsub('[g]', 'X', my.character)
```

```
## [1] "Yes momma I'm a strinX"
```

Datetime data

R stores dates internally as the number of days since the epoch (1 Jan 1970)

```r
my.date <- as.Date("2015-07-27")
my.date + 7
```

```
## [1] "2015-08-03"
```

```
weekdays(my.date + 7)
```

```
## [1] "Monday"
```

```
my.date - 365
```

```
## [1] "2014-07-27"
```

```
weekdays(my.date - 365)
```

```
## [1] "Sunday"
```

the epoch is common to (most) Unix systems

makes it easy to add and subtract days

however, most other languages use seconds since the epoch, not days

these can both cause interoperability issues

Factor data

R stores factors internally as integers, and uses the character strings as labels

```
my.factor <- factor(c('undergraduate','graduate','professor','staff'))
levels(my.factor)
```

```
## [1] "graduate"      "professor"     "staff"         "undergraduate"
```

notice how it sorts those levels alphabetically?

this can cause issues when making plots or trying to display in a particular order - if sort order is critical

try giving your factor explicitly numeric levels and character labels

```
my.factor <- factor(c(1,2,3,4),
                    levels=c(1,2,3,4),
                    labels=c('undergraduate','graduate','professor','staff'))
levels(my.factor)
```

```
## [1] "undergraduate" "graduate"      "professor"     "staff"
```

Testing and changing data types

you can test types with `is.type`, e.g.

```r
is.character(my.character)
```

```
## [1] TRUE
```

```r
is.numeric(my.character)
```

```
## [1] FALSE
```

you can change datatypes with `as.type`, e.g.

```r
as.character(9)
```

```
## [1] "9"
```

```r
as.numeric(my.character)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA
```

trying to coerce types can lead to weird behavior

Data Structures

there are five kinds of data structures in R, but you will probably only ever use three of these

1. vector
2. list
3. dataframe

a vector is an ordered group of the same kind of data, e.g.

```r
my.vector <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
my.vector
```

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

it doesn't matter what the datatype is, as long as it is all the same

```r
your.vector <- c(1,2,3,4,5)
my.vector * your.vector
```

```
## [1] 1 2 0 0 5
```

you will frequently need to create vectors that are sequences of numbers

```
seq(from=0,to=length(my.vector),by=2)
```

```
## [1] 0 2 4
```

R also gives you a shorthand operator for creating sequences where `by=1`

```
1:length(my.vector)
```

```
## [1] 1 2 3 4 5
```

remember what we said about multiplying logical vectors?

you can add and multiply vectors, but they need to be the same length

```
c(1,2,3) * c(TRUE, FALSE)
```

```
## Warning in c(1, 2, 3) * c(TRUE, FALSE): longer object length is not a
## multiple of shorter object length
```

```
## [1] 1 0 3
```

you will run into this issue a bunch dealing with dataframes and logical vectors

you can pull elements out of a vector by

```
my.vector[1]
```

```
## [1] TRUE
```

```
your.vector[1:2]
```

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

```
my.vector[c(1,3)]
```

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

a list is an ordered group of things that are not of the same type

```r
my.list <- list(TRUE, 'two', 3)
my.list
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] "two"
##
## [[3]]
## [1] 3
```

you can find out the attributes for and types of data in a list with

```r
str(my.list)
```

```
## List of 3
##  $ : logi TRUE
##  $ : chr "two"
##  $ : num 3
```

lists are simple containers, and are not additive or multiplicative

```r
my.list * list(1, 'two', FALSE)
```

subsetting a list with brackets pulls out the element along with its attribute

this will be annoying when you try to pull values out of objects like summary(lm())

```r
my.list[1]
```

```
## [[1]]
## [1] TRUE
```

if you want only the element, use double brackets

```r
my.list[[1]]
```

```
## [1] TRUE
```

Data frames

inside R, a dataframe is just a list of equal-length vectors

much like in SQL where a table is a tuple of attributes

```
my.data <- data.frame(n = c(1,2,3),c=c('one','two','three'),b=c(TRUE,TRUE,FALSE))
my.data
```

```
##   n     c     b
## 1 1   one  TRUE
## 2 2   two  TRUE
## 3 3 three FALSE
```

see how this is just a list of vectors?

you can learn some things about data frames

```
dim(my.data) #this gives you nrow() and ncol()
```

```
## [1] 3 3
```

```
colnames(my.data)
```

```
## [1] "n" "c" "b"
```

```
rownames(my.data)
```

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

dataframes have some special operators they share with matrices - subset with brackets

```
my.data[1:2,3]
```

```
## [1] TRUE TRUE
```

dataframes also have special operators that they inherit from lists

```
str(my.data)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ n: num  1 2 3
##  $ c: Factor w/ 3 levels "one","three",..: 1 3 2
##  $ b: logi  TRUE TRUE FALSE
```

```r
my.data$b
```

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

```r
my.data$d <- c(my.date, my.date+7, my.date-7)
my.data
```

```
##   n     c     b          d
## 1 1   one  TRUE 2015-07-27
## 2 2   two  TRUE 2015-08-03
## 3 3 three FALSE 2015-07-20
```

the dollar operator also does partial matching

```r
my.data$really.long.and.complicated.variable.name <- 999
my.data$r
```

```
## [1] 999 999 999
```

since the number of rows in the dataframe (3) is a multiple of the length of the assignment (1), the vectors gets concatenated against itself three times

you can combine data frames with

```r
rbind(my.data, my.data)
```

```
##   n     c     b          d really.long.and.complicated.variable.name
## 1 1   one  TRUE 2015-07-27                                       999
## 2 2   two  TRUE 2015-08-03                                       999
## 3 3 three FALSE 2015-07-20                                       999
## 4 1   one  TRUE 2015-07-27                                       999
## 5 2   two  TRUE 2015-08-03                                       999
## 6 3 three FALSE 2015-07-20                                       999
```

```r
cbind(my.data, my.data)
```

```
##   n     c     b          d really.long.and.complicated.variable.name n
## 1 1   one  TRUE 2015-07-27                                       999 1
## 2 2   two  TRUE 2015-08-03                                       999 2
## 3 3 three FALSE 2015-07-20                                       999 3
##       c     b          d really.long.and.complicated.variable.name
## 1   one  TRUE 2015-07-27                                       999
## 2   two  TRUE 2015-08-03                                       999
## 3 three FALSE 2015-07-20                                       999
```

you'll learn tomorrow about better ways to merge data, especially heterogeneous data

saving console output

introduction

at the end of the day, it's likely that one or two students will want to know how to "save what we did". the commands are of course already in the .R file that the students have been typing their notes into. If they want to save the console output, they basically have three options:

1. copy all the output and paste it into a separate text file; or,
2. use a sink; or,
3. write their notes as .Rmd

sinks

to use a sink, have the student put `sink('filename')` as the very first line in their notes, and `sink()` as the very last. then, when they re-run their entire .R file, the output will go to a pdf called "filename" instead of the R console. for an example, see `save_console_output.*` in the examples directory.

.Rmd

See Dynamic documents in R Markdown

Acknowledgements

Materials taken from:

Hadley Wickham