Week 1: Background, Getting Started, and Nut & Bolts

Katherine Ramírez Cubillos

16/8/2020

R Programming

This document contains all scripts developed in the first week of the R-Programming course of the John Hopkings University, teaching by Rogger D.Peng.

Etering Input

At the R prompt we type expressions. The <- symbol is the assigment operator.

```
x <-1
print(x)
```

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

```
msg <-"hello"
```

The grammar of the language determines whether an expression is complete or not.

y<- ## Incomplete expression

The # character indicates a comment. Anything to the riht of the # (including the # itself) is ignored.

Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

```
x <- 5 ## nothing printed
x      ## auto-printing occurs
```

```
## [1] 5
```

```
print(x) ## explicit printing
```

```
## [1] 5
```

The [1] indicates that x is a vector and 5 is the first element.

Printing

```
x<-1:120
x
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
## [109] 109 110 111 112 113 114 115 116 117 118 119 120
```

The : operator is used to create integer sequences.

Objects

R has five basic or "atomic" classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical(True/Flase)

The most basic object is a vector:

- A vector can only contain objects of the same class
- BUT: The one exception is a list, which is represented as a vector but can contain objects of different classes (indeed, that's usually why we use them)

Empty vectors can be created with the **vector()** function.

Numbers

- Numbers in R a generally treates as numeric objects (i.e. double precision real numbers)
- If you explicity want a integrer, you need to specify the **L** suffix
- Ex: Entering **1** gives you a numeric object; entering **1L**
- There is also a special number **Inf** wich represents infinity; e.g. **1/0; Inf** can be used ordinary calculations; e.g. **1/Inf** is 0
- The value **NaN** represents an undefined value (#not a number"); e.g. 0/0; **NaN** can also be thought of as a missing value (more on that later)

Attributes

R objects can have atrributes:

- names, dimnames
- dimanesion (e.g. matrices, arrays)
- class
- lenght
- other user-defined attributes/metadata

Attributes of n object can be accessed using the **attribites()** function.

Creating Vectors

The **c()** function can be used to create vectots of objects.

```
x<-c(0.5, 0.6)         ## numeric
x<- c(TRUE, FALSE)     ## logical
x<- c(T, F)            ## logical
x<- c("a", "b", "c")   ## character
x<- 9:29               ## integer
x<- c(1+0i, 2+4i)      ## complex
```

Using the **vector** () function

```
x<-vector("numeric", length = 10)
x
```

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

Mixing Objects

What about the following?

```
y <-c(1.7, "a")         ## character
y<- c(TRUE, 2)          ## numeric
y<- c("a", TRUE)        ## character
```

When differente objects are mixed in a vector, coercion occurs so that every element in the vector is of the same calss.

Explicity Coercion

Objects can ve explicity coerced from one class to another using the as.* functions, if available.

```
x<- 0:6
class(x)
```

```
## [1] "integer"
```

```r
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6
```

```r
as.logical(x)
```

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

```r
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

Nonsensical coercion results in **NAS**

```r
x<- c("a", "b", "c")
as.numeric(x)
```

```
## Warning: NAs introducidos por coerción
```

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

```r
as.logical(x)
```

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

```r
as.complex(x)
```

```
## Warning: NAs introducidos por coerción
```

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

Matrices

Marices are vectors with a dimession atribute. The dimession atrribute is itself an integer vector of length 2 (nrow, ncol)

```r
m <- matrix(nrow=2, ncol=3)
m
```

```
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
```

```r
dim(m)
```

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

```r
attributes(m)
```

```
## $dim
## [1] 2 3
```

Matrices are constructed column-wise, so entries can be thought of starting in the "upper left" corner and running down the columns.

```r
m<-matrix(1:6, nrow=2, ncol=3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Matrices can also be created directly from vectors by adding a dimesion atribute

```r
m<-1:10
m
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
dim(m)<-c(2,5)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

cbind-ing and rbind-ing

Matrices can be created by column-binding or row-binding with **cbind()** and **rbind()**

```r
x<-1:3
y<-10:12
cbind(x,y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```r
rbind(x,y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

List

List are a special type of vector that can contain elements of different classes. List are a very important data type in R and you should know them well.

```
x<-list(1, "a", TRUE, 1+4i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

Factors

Factors are used to represent categorical data. Factor can be unordered or ordered. One can think of a factor as an integer vector each integer has a label.

- Factors are treated specially by modelling functions like lm() and glm()
- Using factor with labels is better than using integers because factor are self-describing; having a variable that has values "Male" and "Female" is better than a variable that has value 1 and 2.

```
x<- factor(c("yes", "yes", "yes", "no", "yes", "no"))
x
```

```
## [1] yes yes yes no  yes no
## Levels: no yes
```

```
table(x)
```

```
## x
##  no yes
##   2   4
```

```
unclass(x)
```

```
## [1] 2 2 2 1 2 1
## attr(,"levels")
## [1] "no"  "yes"
```

The order of the levels can be set using the **levels** argument to **factor()**. This can be impportant in linear modelling because the first level is used as the baseline level.

```r
x<-factor(c("yes", "yes", "no", "yes", "no"),
          levels=c("yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: yes no
```

Missing Value

Missing values are denoted by **NA** or **NaN** for undefined mathematical operations.

- **is.na()** is used to test objects if they are **NA**
- **is.nan()** is used to test for **NaN**
- A **NaN** value is also **NA** but the converse is no true.

```r
x<- c(1,2,NA,10,3)
is.na(x)
```

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

```r
is.nan(x)
```

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

```r
x<-c(1,2,NaN, NA, 4)
is.na(x)
```

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

```r
is.nan(x)
```

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

Data Frames

Data frames are used to store tabular data:

- They are represented as a special type of list where every element of the list have the same length.
- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.
- Unlike matrices, data frames can store different classes of objects in each column (just like this); matrices must have every element be the same class.
- Data frames also have a special attribute called **row.names**
- Data frames are usually created by calling **read.table()** or **read.csv()**
- Can be converted to a matrix by calling **data.matrix()**

```
x<-data.frame(foo=1:4, bar=c(T,T,F,F))
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```
nrow(x)
```

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

```
ncol(x)
```

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

Names

R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
x<- 1:3
names(x)
```

```
## NULL
```

```
names(x)<-c("foo", "bar", "norf")
names(x)
```

```
## [1] "foo"  "bar"  "norf"
```

List can also have names

```
x<-list(a=1, b=2, c=3)
x
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

And matrices

```r
m<-matrix(1:4, nrow=2, ncol=2)
dimnames(m)<-list(c("a", "b"), c("c", "d"))
m
```

```
##   c d
## a 1 3
## b 2 4
```

Reading Data

There are few principal functions reading data into R.

- **read.table, read.csv**, for reading tabular data.
- **readLines**, for reading lines of a text file.
- **source**, for reading in R code files (**inverse** of **dump**)
- **dget**, for reading in R code files (**inverse** of **dump**)
- **load**, for reading in saved workspaces
- **unserialize**, for reading single R objects in binary form

Writing Data

There are analogous functions for writing data to files.

- write.table
- writeLines
- dump
- dput
- save
- serialize

Reading Data Files with read.table

The **read.table** function is one of the most commonly used functions for reading data. It has a few important arguments:

- **file**, the name of a file, or a connection.
- **header**, logical indicating if the file has a header line.
- **sep**, a string indicating how the columns are separated.
- **colClasses**, a character vector indicating the calss of each column in the dataset.
- **nrows**, the number of rows in the dataset.
- **comment.char**, a character string indicating the comment character.
- **skip**, the number of lines to skip form the beginning.
- **stringsAsFactors**, should character variables be coded as factors?

Read.table

For small to moderately sized datasets, you can usually call read.table without specifying any other arguments.

```
## data <- read.table("foo.txt)
```

R will automatically:

- Skip lines that begin with a #
- Figure out how many rows there are (and how much memory needs to be allocated)
- Figure what type of variable is in each column of the Telling R all these things directly makes R run faster and more efficently.
- **read.csv** is identical to read.table except that the default separator is a comma.

Reading in Large Dataset with read.table

With much larger datasets, doing the following things will make your life easier and will prevent R from choking.

- Read the help page for read.table, which contains many hints.
- Make a rough calculation of the memory required to store you dataset. If the dataset is larger than the amount of RAN on your computer, you can probably stop right here.
- Set **comment.char = " "** if there are no commented lines in your file.
- Use the **colClasses** argument. Specifying this option inStead of using the default can make 'read.table' run much faster, often twice as fast. In order to use the option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set **colClasses="numeric"**, A quick an dirty way to figurre out the classes of each column is the following:

```
## initial <- read.table("datatable.txt", nrows = 100)
## classes <- sapply(initial, class)
## tabAll <- read.table("datatable.txt",
## colClasses = classes)
```

Set **nrows**. This doesn't make R run faster it helps with memory usage. A mild overestimate is okay. You can use the Unix tool **wc** to calculate the number of lines in a file.

Known The System

In general, when using R with larger datasets, it's useful to know a few things about your system:

- How much memory is available?
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system?
- Is the OS 32 or 64?

Memory Requirements for R objects

I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame?

1,500,000 $\times$ 120 $\times$ 8 bytes/numeric

= 1440000000 bytes

= 1440000000 / bytes/MB

= 1,373.29 MB

= 1.34 GB

Textual Formats

- **Dumping and dputing** are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- **Unlike** writing out a table or cvs file, **dump** and **dput** preserve the *metadata* (sacrificing some readability), so that another user doesn't have to specify it all over again.
- **Textual** formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files.
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problbem.
- Textual formats adhere to the "Unix philosophy"
- Downside: The format is not very space-efficient.

dput-ting R Objects

Anotherway to pass data around is by deparsing the R object with dput and reading it back in using **dget**

```
y <- data.frame(a = 1, b = "a")
dput(y)
```

```
## structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), class = "data.frame", row.n
## -1L))
```

```
dput(y, file = "y.R")
new.y <- dget("y.R")
new.y
```

```
##   a b
## 1 1 a
```

dumping R Objects

Multiple objects can be deparsed using the dump function and read back in using **source**.

```
x<-"foo"
y<- data.frame(a=1, b="a")
dump(c("x", "y"), file="data.R")
rm(x,y)
source("data.R")
y
```

```
##   a b
## 1 1 a
```

```
x
```

```
## [1] "foo"
```

Connections: Interfaces to the Outside World

Data are read in using connection interfaces. Connections can be made to files (most common) or to other more exotic things.

- **File**, opens a connection to a file.
- **Gzfile**, opnes a connection to a file compressed with gzip.
- **BZfile**, opnes a connection to a file compressed with bzip2.
- **url**, opens a connection to a webpage.

File Connections:

```
str(file)
```

```
## function (description = "", open = "", blocking = TRUE, encoding = getOption("encoding"),
##     raw = FALSE, method = getOption("url.method", "default"))
```

- Description is the name of the file.
- Open is a code indicating:
- "r" read only
- "w" writing (and iniatilizing a new file)
- "a" appending -"rb", "wb", "ab" reading, writing, or appending in binary mode (Windows)

Connections:

In general, connections are powerful tools that let you navigate file or other external objects. In practice, we often don't need to deal with the connection interface directly.

```
## con <-file("foo.txt", "r")
## data<- read.csv(con)
## close(con)
```

is the same as

```
## data<- read.csv("foo.txt)
```

Reading lines of a text file

writeLines takes a character vector and writes each element one line at a time to text file.

```
## con <-gzfile("words.gz")
## x<-readLines(con, 10)
## x
```

readLines can be useful for reading in lines of webpages

```
## This might take time
con <- url("http://www.jhsph.edu", "r")
x <- readLines(con)
head(x)
```

```
## [1] "<!DOCTYPE html>"
## [2] "<html lang=\"en\">"
## [3] ""
## [4] "<head>"
## [5] "<meta charset=\"utf-8\" />"
## [6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"
```

Subsetting

There are a number of operators that can be used to extract subsets of R objects.

- [always returns an object of the same class as the original; can be used to select more that one element (there is one exception)
- [[is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
- **$** is used to extract elements of a list or data frame by name; semantics are similar to that of [[.

```
x<- c("a", "b", "c", "c", "d", "a")
x[1]
```

```
## [1] "a"
```

```
x[2]
```

```
## [1] "b"
```

```
x[1:4]
```

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

```
x[x>"a"]
```

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

```
u<- x > "a" ##logical vector
u
```

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

```
x[u]
```

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

Subsetting List

```
## Example 1
y<- list(foo=1:4, bar=0.6)
y[1]
```

```
## $foo
## [1] 1 2 3 4
```

```
y[[1]]
```

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

```
y$bar
```

```
## [1] 0.6
```

```
y[["bar"]]
```

```
## [1] 0.6
```

```
## Example 2
z<- list(foo=1:4, bar=0.6, baz="hello")
z[c(1,3)]
```

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

The [[operator can be used with computed indices; $ can only be used with literal names.

```
v<-list(foo=1:4, bar=0.6, baz="hello")
name <- "foo"
v[[name]]  ## compued index for 'foo'
```

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

```
v$name     ## element 'name' doesn't exist!
```

```
## NULL
```

```
v$foo      ## element 'foo' does exist
```

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

The [[can take an integer sequence

```
b<-list(a=list(10,12,14), b=c(3.14, 2.81))
b[[c(1,3)]]
```

```
## [1] 14
```

```
b[[1]][[3]]
```

```
## [1] 14
```

```
b[[c(2,1)]]
```

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

Subsetting a Matrix

Matrices can be subsetted in the usual with *(i,j)* type indices.

```
x<-matrix(1:6, 2,3)
x[1,2]
```

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

```
x[2,1]
```

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

Indices can also be missing

```
x[1,]
```

```
## [1] 1 3 5
```

```
x[,2]
```

```
## [1] 3 4
```

By default, when a single element of a matriz is retrieved, it is returned as a vector of length 1 rather than a 1x1 matrix. This behavior can be turned off by setting **drop="False"

```
x<-matrix(1:6,2,3)
x[1,2]
```

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

```
x[1,3, drop=FALSE]
```

```
##      [,1]
## [1,]    5
```

Similary, subsetting a single column or a single row will give you a vector, not a matrix (by default)

```r
x<-matrix(1:6, 2,3)
x[1,]
```

```
## [1] 1 3 5
```

```r
x[1, , drop=FALSE]
```

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

Partial Martching

Partial marching of names is allowed with [[**$**.

```r
x<- list(aardvark=1:5)
x$a
```

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

```r
x[["a"]]  ##By default [[]] doesn't do partial matching like dollar sign does. It gets null back, becau
```

```
## NULL
```

```r
x[["a", exact=FALSE]]
```

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

Removing NA values

A common task is to remove missing values **(NAs)**

```r
x<- c(1,2,NA,4,NA,5)
bad <-is.na(x)  ## logical vector which is true if the element is missing and false if it's not missing
x[!bad]
```

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

What if there are multiple things and you want to take the subset with no missing value?

```r
x<-c(1,2,NA,4,NA,5)
y<-c("a", "b", NA, "d", NA, "f")
good<-complete.cases(x,y)
good
```

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

```
x[good]
```

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

```
y[good]
```

```
## [1] "a" "b" "d" "f"
```

It's possible revoe missing values from data frames.

```
airquality[1:6,]
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```
good <-complete.cases(airquality)
airquality[good, ][1:6, ]
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 7    23     299  8.6   65     5   7
## 8    19      99 13.8   59     5   8
```

Vectorized Operations

Many operations in R are *vectorized* making code more efficient, concise, and easier to read:

```
x<-1:4; y<-6:9
x+y
```

```
## [1]  7  9 11 13
```

```
x>2
```

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

```
x>=2
```

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

```r
y==8
```

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

```r
x*y
```

```
## [1]  6 14 24 36
```

```r
x/y
```

```
## [1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Vectorized Matrix Operations

```r
x<-matrix(1:4, 2,2); y <-matrix(rep(10,4), 2,2)
x*y    ## element-wise multiplication
```

```
##      [,1] [,2]
## [1,]   10   30
## [2,]   20   40
```

```r
x/y
```

```
##      [,1] [,2]
## [1,]  0.1  0.3
## [2,]  0.2  0.4
```

```r
x%*%y   ## true matrix multiplication
```

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