R Fundamentals - Class # 1

J. Primavera

October 12, 2016

Tips

- These slides mix plain text and R code (grey background)
- Everything to the right of "#" symbol is a comment

```
# this is a comment
x <- 1
y <- 2  # this is also a comment
```

- It is recommended to replicate the code in these slides while we scroll them to get familiar with R syntax
- R is case sensitive!
- Press CTRL+Enter to run programs from RStudio script editor
- To pull up documentation for a function run ?*namefunction*
- Remember R does not like back slash in filepaths

R & your system

Working directory

Once R is installed in your computer it is able to communicate and interact with your system (create folders, read existing files, etc.)

First, let's see where we are

```
getwd()
```

getwd() is a function without arguments returning the filepath to your current working directory

Working directory is the place which by default R communicate with (save and load file, etc.)

Change working directory

Create a folder called “RFundamentalsWeek1” with **dir.create()**

```
dir.create("C:/Users/pc/Desktop/RFundamentalsWeek1")
```

and set it as your working directory with **setwd()**

```
setwd("C:/Users/pc/Desktop/RFundamentalsWeek1")
```

We said that working directory is the R default *interaction* folder with your system, then guess what this will produce:

```
dir.create("sub")
```

. . . a sub-folder in your working directory

Check content of folders

Check what is inside your working directory with **dir()**

```
dir()  # can you see "sub"?
```

- dir() search in your WD because no other path is specified
- But you can check any folder in your system

```
dir("C:/Users")
```

- Shortcuts "." and ".." help you navigate in your system

```
dir("./sub")  # "." set the start in your WD
dir("..")  # ".." moves you one level up
```

R workspace

Workspace is the collection of all objects created during an R session

list all objects in your workspace with **ls()** function

```
ls()  # character(0) indicates empty
```

- Create your first object named "x" taking value 1

```
x <- 1
```

- Assignment operator "<-" is used to create obects in R
- Top-right box in RStudio represents your working space (you should see "x" now)
- re-runnig **ls()** now should return "x" object

Remove objects from workspace

Let's create a bunch of objects:

```
y <- 99; msg <- "Hello"; msg2 <- "Hi"
```

Now let's remove "x" from the workspace with **rm()** function

```
rm("x")
```

concatenating **rm()** and **ls()** we can clean-up all workspace

```
rm(list=ls())  # In R is very common to nest functions
```

To understand why we used *list* argument read documentation

```
?rm
```

R basic obejcts and operators

Objects' classes in R

In R there are four important classes of objects:

```
"Hola"     # character, any string within quotes
3.14        # numeric, any real number
4L       # integer, any integer number
TRUE     # logical, TRUE or FALSE reserved words
```

Check the class of these objects with function **class()**

```
class("Hello")
class(3.14)
class(4L)
class(4) # without suffix "L" all numbers are numeric by de
class(TRUE)
```

Arithmetic operators

given two numeric objects R can perform the most common arithmetic operations:

```
3 + 4
3 - 4
3 * 4
3 / 4
abs(3 - 4)
3^4   # or 3**4
sqrt(4)
```

In R expressions are directly evaluated and the result is returned to the console

logical operators

- given a couple of atomic objects R can perform logical operations
- logical operations return a logical value (TRUE, FALSE)

```
3 == 4  # equality
"a" == "a"
3 > 4  # greater than
3 <= 4  # lower or equal than
3 != 4  # different from
"hello" == "Hello"
```

which can be combined using AND (&) and OR (|) operators

```
4 >= 3 & 3==3
4 < 3 | 3==3
```

Atomic vectors

The simplest data structure

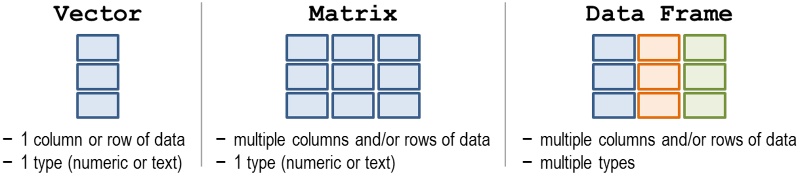
- Vectors represent the simplest data structure in R
- Even single-elements objects are seen as vectors (of length one)

```
length("Hello")
length(2)
length(TRUE)
```

- That's why we call vectors atomic vectors
- A vector is a collection of elements all of the same class (character, logical, etc.)

More complex data structures

- More complex data structures can be seen as extensions of vectors



- More on these structures later. . .

Create vectors with combine function

- Create vectors of length>1 with **c()** function

```
c("Hola", "Ciao", "Hello", "Bonjour")  # character vector
c(0.99, 2.4, 1.4, 5.9)  # numeric vector
c(1L, 2L, 3L, 4L)  # integer vector
c(TRUE, TRUE, FALSE, TRUE)  # logical vector
```

Check their class:

```
class(c("Hola", "Ciao", "Hello", "Bonjour"))
class(c(0.99, 2.4, 1.4, 5.9))
class(c(1L, 2L, 3L, 4L))
class(c(TRUE, TRUE, FALSE, TRUE))
```

Other ways to create vectors

Create integer vectors with **seq()** function (or ":" operator)

the following four expressions all produce the same result:

```
seq(from = 1, to = 4, by = 1)
seq(from=1, to=4)      # by=1 is default
seq(1, 4)       # arguments in R can be matched by position
1:4        # common operations in R have shortcuts
```

Create vectors using **rep()** function

```
rep(x = "a", times = 4)  # replicate "a" four times
rep("a", 4)  # same as above
rep(c("a", "b"), times = 2)  # same but for a vector
rep(c("a", "b"), each = 2)  # element-by-element
```

Subsetting vectors

- [logical index]

```
x <- 1:10
x >= 5
idx <- (x > 5)
x[idx]  # all values of x greater then 5
x[x < 7] # calculate index directly within brackets
```

- [positive integers index]

```
x[1]     # 1st element
x[c(1,5)]    # 1st and 5th element
```

- [negative integers vector]

```
x[-1]    # all but the 1st
x[-c(1,10)]    # all but the 1st and the 10th
```

Arithmetic and logical operators are vectorized

- we say that a function is vectorized when it works both on vectors (and matrices) and scalars

What do you expect these expressions will return?

```
c(1, 2, 3, 4) + c(5, 6, 7, 8)
c(1, 2, 3, 4) / c(5, 6, 7, 8)
sqrt(c(1, 2, 3, 4))

c(1, 2, 3, 4) == c(5, 6, 7, 8)
c(1, 2, 3, 4) != c(5, 6, 7, 8)
```

- R perform the operation element-by-element and return the vector of results so obtained
- Keep in mind that most funcions in R are vectorized. . .

Vectorization + Recycling

- we saw operations between vectors of same length:

```
c(1, 2, 3) + c(5, 6, 7)  # simple element-by-element
```

- but what if length differs?
- In the case when one is multiple of another:

```
c(1, 2) + c(5, 6, 7, 8)  # shortest vector "recycled"
c(1, 2, 1, 2) + c(5, 6, 7, 8)
```

- The case when one isn't multiple of another

```
c(1, 2) + c(5, 6, 7)  # recycling + warning
r <- c(1, 2, 1) + c(5, 6, 7)
```

Useful functions for numerical objects

- summarizing a numerical vector

```
mynum <- c(3.14, 6, 8.99, 10.21, 10, 56.9, 32.1, 2.3)
sum(mynum)
mean(mynum)
sd(mynum)    # standard deviation
median(mynum)
```

- what if we want the skewness of this vector?
- We could check the formula and write our own function or we could search the internet (google this “skeweness function in r”)

Install a package and use its functions

First Google result mentions a R package called *e1071*

```
install.packages("e1071")  # install the package
library(e1071)  # load the package
```

Now all the functions in this package are available to use:

```
skewness(mynum)
kurtosis(mynum)
```

There are almost 10.000 packages in CRAN (and many others off-CRAN), so just type your problem in google with an *R* tag and odds are you will find a built-in solution in some package

Useful functions for logical objects

- underlying structure of logical values is TRUE=1 and FALSE=0

```
mylogic <- c(F, T, F, rep(T, 3))
sum(mylogic)
```

- obtain the TRUE indices of a logical object with **which()**

```
which(mylogic)
```

- summarizing logical vectors...

```
any(mylogic)  # is at least one of the values TRUE?
all(mylogic)  # are all of the values TRUE?
```

Useful functions for character objects

```
mychar <- c("201510", "201511", "201512", "201601")
# the ubiquitous substring...
substr(x = mychar, start = 1, stop = 4)
nchar("Hello")     # number of characters in a string
```

- concatenate character vectors

```
paste("I", "m", sep = "'")
paste("N.", 1, sep="")    # 1 is coerced to "1"
```

- find and replace

```
gsub(pattern = "20", replacement = "", x = mychar)
```

Implicit coercion

Coercion happens when we force an object to belong to a class

- implicit coercion numeric vs CHARACTER

```
c(1.7, "a")
class(c(1.7, "a"))
```

- implicit coercion logical vs NUMERICAL

```
c(FALSE, 2)
class(c(TRUE, 2))
```

- implicit coercion CHARACTER vs logical

```
c("a", TRUE)
class(c("a", TRUE))
```

Holds a principle of least common denominator. . .

Explicit coercion

Family of functions of form **as.*** coerce explicitly R objects

- consider the following numeric vector

```
x <- c(0, 1, 2, 3, 4, 5, 6)
class(x)
```

Force it to a character or logical (what do you expect to happen?)

```
as.character(x)
as.logical(x) # 0=FALSE, 1+ = TRUE
```

non-sensical coercion returns missing values:

```
as.numeric(c("a", "b", "c"))
as.logical(c("a", "b", "c"))
```

Special values in R

Missing values

- **NA** is a reserved word in R indicating a missing value
- reserved words have special meaning and cannot be used as identifier (variable name, function name, etc.)

```
NA <- 1  # This will trigger an error!
```

- You can use the **NA**s to create a placeholder for a value that exist but you don't know...

```
year <- c(2012, 2013, 2014)
gwp <- c(NA, 98.7, 32.5)
```

- class deduced from non-missing elements

```
class(gwp)
is.na(gwp)  # indicates which elements are missing
```

Other special values

For a list of reserved words in R type this:

```r
help(reserved)
```

- **NULL**, represents an object which is absent

```r
x <- NULL    # useful to initialize objects to be filled la
```

- **Inf**, -**Inf**, **NaN** (special words for mathematical concepts)

```r
1/0  # infinite
-1/0  # minus infinite
0/0  # undefined number
```

(bear in mind that NaN is also NA, viceversa is not true)

Matrices

Matrx underlying structure

- In R matrices can be seen as vectors with a dimension attribute
- To highlight this idea let's create a matrix in a not-so-common way:

```
x <- 1:6  # take a vector
dim(x)    # vector do not have dimension attribute
dim(x) <- c(2, 3)   # impose a 2x3 dimesion (2 rows, 3 col
class(x)  # here it is a matrix!
x
```

- This tricky way to create a matrix is not so common, but it is useful to understand the underlying structure of objects in R. . .
- . . . and so be able to better manipulate them to our needs)

More common ways to create matrices

- with function **matrix()**

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

- by binding rows or columns with functions **rbind()** or **cbind()**

```
x <- 1:3
y <- 10:12
m1 <- cbind(x,y)
m2 <- rbind(x,y)
class(m1)
class(m2)
```

Subsetting matrices

- Matrices can be subset using (i,j)-style index

```
m[1,2]   # one single element
m[1,]    # one full row
m[,3]    # one full column
m[,-1]   # all columns but one
```

- Can you think about another way to obtain the last result?
- *Tip*: use an integer vector with function **c()**

Factors

Nominal factors

- Factors are used to describe items that can have a finite number of values (i.e. categories)
- You can see them as positive-integer-sequences with labels

```
f <- factor( c("f", "m", "m", "f", "f") )
class(f)
```

- Factors have a *levels* attribute listing its unique categories
- Access levels attribute with **levels()** function

```
attributes(f)
levels(f)
```

Ordered factors

If a factor has a natural order this should be specified

```
fo <- factor( c("low", "med", "low", "high"), ordered = TRU
```

- Default order is alphabetical

```
levels(fo) <- c("low", "med", "high")  # re-order
```

- It can useful sometimes re-order also nominal factors (e.g. to change default base levels taken by a GLM)

```
levels(f) <- c("m", "f")     # change alphabetical default
```

- Obtain frequency count of factor combinations with **table()**

```
table(f)
```

Data frames

Create a data frame from scratch

- R structure which most closely mimic SAS data set (i.e. a "cases by variables" matrix of data)
- R-speaking, it is a collection of vectors and factors all having the same length
- A data frame generally has *names* and *row.names* attributes to label variables and observations respectively
- You create a data.frame with function **data.frame()**

```
df <- data.frame( x = 1:3, y = c("a", "b", "c"),
                  f = factor( c("m", "f", "m") ) )
class(df)
```

- Although more often you will create a data.frame by *reading* some data from a file (excel, internet, SAS, etc.)

Read some insurance data

- Copy “PolicyPtfSum.csv” in your working directory
- This is an anonymized summarized policy portfolio
- Contains TPL policies and Material/BI claims
- Original data is exmaple dataset of AXAML package

Read a csv file with function **read.csv()** (more on session #3)

```
ptf <- read.csv("PolicyPtfSum.csv")
```

Have an overview of the data using these functions

```
str(ptf)    # returns a compact summary of R objects
summary(ptf)    # few statistics for each variable
head(ptf, n = 20)   # visualize first 20 observations
tail(ptf)   # last 6 observations
```

Subset data frames

[i,j]-index notation is valid also for data.frames

```
ptf[1,1]
ptf[1,5]
```

Additionally you can retain one or more variables by name

```
ptf$Occupation  # using $ operator
ptf[, "Occupation"]  # quoting variable's name in j slot
ptf[, c("Occupation", "Gender")]
```

Tip: after you type "$" wait for RStudio auto-completion menu

Tip: In general press "Tab" to ask RStudio auto-completion options

Analyse data frames

Use the **sum()** function to get some overall statistics from this data

```
sum(ptf$exposure)    # tot exposure
sum(ptf$bclaim) / sum(ptf$exposure)     # avg BI freq
sum(ptf$mcost) / sum(ptf$mclaim)    # avg BI sev
```

Calculate same statistics only for males:

```
sum(  ptf[ ptf$Gender=="Male", "exposure" ]   )
sum( ptf[ ptf$Gender=="Male", "bclaim" ] ) /
  sum(  ptf[ ptf$Gender=="Male", "exposure" ]  )
```

Useful functions to analyse data frames

You can see that syntax become twisted quite rapidly when more complex manipulation is needed (filter rows, select columns, etc.)

Use **with()** and **subset()** to make your program more readable

```r
# with() allows to call dataframe's variables directly
with( ptf, sum(bclaim)/sum(exposure) )

#subset() returns a dataframe meeting certain conditions
subset( ptf, Gender=="Male" )
```

Recalculate male average BI claim frequency:

```r
with( subset(ptf, Gender=="Male"),
      sum(bclaim)/sum(exposure)
      )
```

Subset data frames to remove missing values

Control if there is some missing value with **is.na()** function

```
sum( is.na(ptf) )
```

Understand which variables have missing values with **which()** function with **arr.ind = TRUE** argument

```
class(is.na(ptf))
w <- which( x = is.na(ptf), arr.ind = TRUE )
head(w)
unique(w[,2])  # all missing values are in 4th variable
```

Note: When x has dimesion > 1 then the *arr.ind* argument tells R whether array indices should be returned

Now subset the dataframe by eliminating rows where *occupation_na* is missing:

```
ptf_clean <- subset(ptf, !is.na(Occupation_na))
```

A method to eliminate all records including at least one missing value (no matter in which variable) is with function **complete.cases()**

It returns a logical vector indicating which cases (full record) are complete

```
good <- complete.cases(ptf)  # "good" is a logical vector
ptf_clean <- ptf[good,]
```

Add new variables to a data frame

Add a variable with random values 1:10 using **sample()** function

```
ptf$new <- sample(1:10, nrow(ptf),
                        replace = TRUE)
```

Concatenate the values of two variables to create an interaction term

Use the **paste()** function

```
ptf$interac <- with(ptf,
                    paste(Gender, Category_Car, sep="_"))

class(ptf$interac)    # indeed it's not factor
ptf$interac <- as.factor(ptf$interac)    # coerce it
table(ptf$interac)
```

Calculating an index of association

- V-Cramer is an association index ranging from 0 to 1
- I found a piece of code defining the function here

```
# install.packages("vcd")
catcor <- function(x, type=c("cramer", "phi",
                              "contingency")) {
  require(vcd)
  nc <- ncol(x)
  v <- expand.grid(1:nc, 1:nc)
  type <- match.arg(type)
  res <- matrix(mapply(function(i1, i2) assocstats(
    table(x[,i1],
          x[,i2]))[[type]], v[,1], v[,2]), nc, nc)
  rownames(res) <- colnames(res) <- colnames(x)
  res
}
```

- More on how to build your own functions in the next class. . .
- For now let's just enjoy the sharing philosophy of R community

Once you run the function definition you can call the function:

```
catcor(x = ptf_clean[,c("Gender",
                        "Category_Car",
                        "Occupation"
                        )],
       type = "cramer")
```

In the next class

Class # 2

We will move from an *interactive* to a **developing** way of programming. . .

Some topics we will go through:

- Lists
- Control structures (if, looping, etc.)
- Writing functions (syntax, arguments, etc.)
- Functions evaluation (arguments matching, etc.)
- Exception handling (stop, error, warning, etc.)
- Probability distributions (rnorm, pnorm, runif, etc.)
- Other