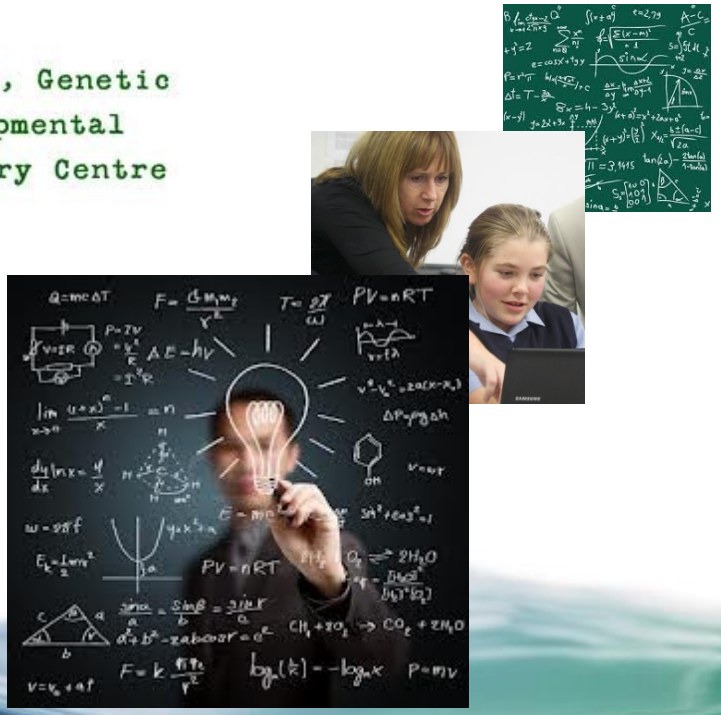
Introduction to R
Data Structure
June 16 2014

Introduction to R: Data structure

Karim Malki

June 16 2014

MRC Social, Genetic & Developmental Psychiatry Centre



Objectives

Data Structure

- Understand Vectors, Lists, Matrices, Attributes and DataFrames
- Understand indexing and how to extract elements from Vectors and DataFrames
- Know how to access objects, subset and work with your data
- Understand how R deals with missing data
- Gain an appreciation of the “recycling rule”

Overview A note about R

Statistics vs Signal processing:

- R is a predominantly statistical software as opposed to a numerical computational software
  - Provides functions for non-trivial statistical operations
  - Can be used for signal processing (eg Fourrier transformations) but it is not very efficient.

- R is an interpreted language:

  - Advantage: less time writing code
  - Disadvantage: computations can be slower
    - But! For most uses, it can handle large data well and reasonably fast

- It is a real programming language: we are not crucified by the developers and are free to use it in a way that mirrors our ideas and imagination.

  - As with every programming language there is always more than one way to do things and never a “right way”.
  - R: it tends to load all the data into memory (workarounds with R version 3).
  - Language of Choice for prototyping – can access databases

Erm..Windows? Don’t take my word for it

```
library(fortunes)
fortune("install")
```

Benjamin Lloyd-Hughes: Has anyone had any joy getting the rgdal package to compile under windows?
Roger Bivand: The closest anyone has got so far is Hisaji Ono, who used MSYS (http://www.mingw.org/) to build PROJ.4 and GDAL (GDAL depends on PROJ.4, PROJ.4 needs a PATH to metadata files for projection and transformation), and then hand-pasted the paths to the GDAL headers and library into src/Makevars, running Rcmd INSTALL rgdal at the Windows command prompt as usual. All of this can be repeated, but is not portable, and does not suit the very valuable standard binary package build system for Windows. Roughly: [points 1 to 5 etc omitted]
Barry Rowlingson: At some point the complexity of installing things like this for Windows will cross the complexity of installing Linux... (PS excepting live-Linux installs like Knoppix)

Data Structures Tables and Numbers:

How many kind of Tables?

- Vectors (tables of dimension 1),
- Matrices (tables of dimension 2),
- Arrays (tables of any dimension),
- Data Frames
- (tables of dimension 2, in which each column may contain different type of data – eg. with one row per subject and one column per variable)

Vectors: Dimension 1

The basic data structure in R is the vector and are of two types: atomic vectors and lists.

They have three common properties:

- Type, typeof(), what it is (there are types logical, integer, double (often called numeric), character. complex and raw.
- Length, length(), how many elements it contains.
- Attributes, attributes(), additional arbitrary metadata.

#create your first (Atomic) Vector (c stands for concatenate or combine)

```
c(1,2,3,4,5)
[1] 1 2 3 4 5
```

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

```
seq(1, 5, by=1)
[1] 1 2 3 4 5
```

```
scan()
[1] 1 2 3 4 5
```

Vectors: Selecting elements:

#create an object vector called “data” containing numbers from -2 to 2 in .1 increments.

```
data <- seq(-2, 2, by=.1)
data
 [1] -2.0 -1.9 -1.8 -1.7 -1.6 -1.5 -1.4 -1.3 -1.2 -1.1 -1.0 -0.9 -0.8 -0.7 -0.6
[16] -0.5 -0.4 -0.3 -0.2 -0.1  0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9
[31]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0
ls()
“data”   [ to remove your object use rm(data)]
str(data)
```

Vectors Indexing

```
data <- seq(-2, 2, by=.1) #recreate the deleted object if you have deleted it
ls();str(data)
data
[1] -2.0 -1.9 -1.8 -1.7 -1.6 -1.5 -1.4 -1.3 -1.2 -1.1 -1.0 -0.9 -0.8 -0.7 -0.6
[16] -0.5 -0.4 -0.3 -0.2 -0.1  0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9
[31]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0

#extract element 5 to 10
data[5:10]
[1] -1.6 -1.5 -1.4 -1.3 -1.2 -1.1

#extract element 2 and 14 to 19
data[c(2,14:19)]
[1] -1.9 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.6 -0.4 -0.3 -0.2 -0.1
[1] -1.6

#create a new vector without elements 19 through 22
new<-data[-(19:22)]
new;length(data);length(new)
```

Vectors Indexing

```
#Find which elements are positive (larger than 0)
data>0
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
[25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[37]  TRUE  TRUE  TRUE  TRUE  TRUE

#Return elements that are positive (greater than 0)
data[data>0]
[1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
[20] 2.0

#create a shorter vector from data
short<-data[1:10]
names(short)
NULL

#add names to your elements
letters
names(short)<-letters[1:length(short)]

short["e"]
e
-1.6
```

Vectors Word of caution

- All elements in an Atomic vector must be of the same type!
  - If you need to have different types of elements within a vector then you must use lists (more on this later).

Generally you test what any object is, whether it is a vector, matrix, dataframe, etc.
However is.vector() does not test if the object is actually a vector and only returns a TRUE statement is the vector has no attributes (again, more on attributes in a little while).

```
#To test if an object is indeed a vector use is.atomic() || is.list()

P<-c(1,2,3,4,5)
is.atomic(P) || is.list(P)  # “||” means OR
```

```
[1] TRUE
```

Data Structure Factors:

A factor is a vector coding for a qualitative variable. For example: colour or gender or where numeric values do not have meaning such as with zip codes.

?factor

```
#create a vector of factors
food <- factor( sample(c("pizza", "pasta", "ravioli"), 10, replace=T) )

#take a look and check that it is indeed a factor
food
class(food)

g <-c("pizza", "pasta", "ravioli")

x <- factor( sample(food, 5, replace=T), levels=g )

levels(x)

table(x)
```

```
g
pizza   pasta ravioli
    3       1       1
```

Data Structure Factors:

```
#Create a Factor object using a dataframe
x <- c("A", "B", "C ")
y <- 1:2
z <- c("male", "female")

expand.grid(x,y,z)

data<-expand.grid(x,y,z)   #This is now a dataframe (more about this later)
is.data.frame(data)
class(data)
names(data)
str(data)
#rename your factors something sensible
names(data)<-c("group", "order", "gender")
Data

#Acess third row in row “group, will return the value and the possible levels
data$group[3]
[1] C
Levels: A B C
```

Data Structure Dataframes:

Broadly it is a list of vectors of the same length. What is special about a dataframe is that they can contain both quantitative(numbers; each column may contain a measurement in a different unit) and qualitative (strings or factors) variables.

```
#Create your first dataframe
n <- 10
df <- data.frame( x=rnorm(n), y=sample(c(T,F), n, replace=T) )
df

class(df)
[1] "data.frame”

dim(df)
[1] 10  2  #Remember it is always rows first and then columns

names(df)
[1] "x" "y"
```

Data Structure Dataframes:

```
summary(df)
 x                y
 Min.   :-1.84062   Mode :logical
 1st Qu.:-0.61141   FALSE:7
 Median : 0.01757   TRUE :3
 Mean   :-0.04692   NA's :0
 3rd Qu.: 0.49319
 Max.   : 1.75365

#Change name of your columns:

names(df) <- c("Z", "Case")
names(df)
[1] "Z"    "Case”

na<-letters[1:10]
row.names(df)<-na
row.names(df)
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j”
str(df)
```

```
'data.frame':    10 obs. of  2 variables:
 $ Z   : num  0.6805 0.4936 0.0831 -0.8832
-0.5778 ...
 $ Case: logi  FALSE TRUE FALSE TRUE TRUE
FALSE ...
```

Access Dataframes:

Use the $ to access a specific column by variable name:

#To access column “Z” or “Case” we simply write

df$Z # access column Z
df$Case # access column Case

an alternative way is to use indexing

To access column “Z” or “Case” we simply use

df[,1] #This reads: access data frame df, the blank before the coma means “all rows” and the “1” means row 1. Remember Rows-Columns

df[,2] #Accesses column 2 “Case”

#to access the element in the second row of the first column – the two ways should give the same answer
df[2,1]
df["b","Z"]

Access Dataframes:

#To access row 4, column 1 in three different ways, simply:

```
df[4,1]
df$Z[4]
df["d","Z"]
```

#To access the elements in rows 2 to 5 for both variables in your dataframe:

```
df[2:5,]
df[2:5,1:2]  #exactly the same as above
```

Access Dataframes:

Attach & Detach

#Columns in a dataframe can be temporarily turned into actual variables to facilitate access to elements with the attach command (this is borrowed from the namespaces in C++).

#Access one of R datasets, in this case observations from an old geyser.

```
?faithful
data<-faithful
names(data)
dim(data)
class(data)
str(data)

attach(data)
str(eruptions)
```

Access Dataframes:

Attach & Detach

#Access the 5th value in the column eruptions

```
eruptions[5]
```

#Test the strength of the association between eruption and waiting time

```
cor.test(data$eruptions,data$waiting)
cor.test(eruptions,waiting)
```

Remember to detach

```
detach()
```

Subsetting data:

Use another built in dataset called “Orange”

```
data(Orange)
data(Orange)

#explore you data
class(Orange);dim(Orange);names(Orange);str(Orange)


#Use the subset command
#Select only Tree 1
dim(Orange)
Orange
d<-subset(Orange, Tree==1)   #”==“stands for “exactly equal”
dim(d)
d
```

Subsetting data:

subset and select

#create a new dataframe with only Tree and Age columns for those trees with a circumference less than 150

```
d1<-subset(Orange, circumference<150, select = c(Tree, age))
dim(d1)
str(d1)
d1

#Do you need the subset command? No, it is just a fancy wrapper.

#You can do the same with indexing

c<-Orange[Orange$circumference<150,1:2]
dim(c)
c
```

Missing data:

Traditionally R codes missing values as Not Available ,"NA".

```
#Create a Vector with a missing value
a<-c(1:5,NA,7:10)
a
str(a)
length(a)


#find the mean of a
mean(a)
[1] NA
```

You need to tell R to ignore the missing value

```
mean(a, na.rm=T)
a1<-mean(a, na.rm=T)
a1
```

Missing data:

#In fact you you could remove NA's yourself with the na.omit function

```
b<-na.omit(a)
b
[1]  1  2  3  4  5  7  8  9 10
attr(,"na.action")
[1] 6
attr(,"class")
[1] "omit"
```

You can use this with dataframes too.

```
#Create a dataframe called d
d <- data.frame(a, b=rev(a))
#check d
d      #omit all rows that contain at least one NA.
e<-na.omit(d)
#Look at e, you will see all NA's removed
e
```

Missing data:

Important! Do not use “NA” in a boolean test.

```
#Use your vector with missing data “a”
a == 8

#Now try with NA
a==NA
[1] NA NA NA NA NA NA NA NA NA NA

#To test for missing values use is.na function
is.na(a)
```

Data Structure: Matrices

- Matrices are 2-dimensional tables.
- They are different from dataframes in that their elements all have to be of the same type. R handles matrices reasonably well which makes it an ideal open-source platform for the creation of mathematical instruments that rely on matrix algebra. Other programs may be better (eg Fortran) but have their own disadvantages.

Data Structure: Matrices

```
#Create your first matrix
    m <- matrix( c(1,2,3,4), nrow=2 )

#check if indeed you have created a matrix
    is.matrix(m)

#Create a larger matrix
    matrix( 1:3, nrow=3, ncol=3 )
```

#Note! elements of a matrix are represented vertically, you can specify it the other way either by transposing the matrix or telling R in the first place

```
    matrix( 1:3, nrow=3, ncol=3)
```

Data Structure: Matrices

```
matrix( 1:3, nrow=3, ncol=3, byrow=T )
or
t(matrix( 1:3, nrow=3, ncol=3 ))
```

- You can perform matrix algebra easily and manipulate your matrices as any other object. R has also some inbuilt functions that make working with matrices easier (or add functionality using the “Matrix” package)

```
#Create two matrices and try adding, subtracting, multiplying, transposing
c1<-cbind( c(1,2), c(3,4));
c2<-rbind( c(1,3), c(2,4))
class(c1);class(c2)
c1*c2
```

You can index elements in a matrix using the same indexing techniques we have look at previously

Data Structure: Lists

#Lists are useful when we need to store complex data as they can contain anything.

#Let’s clear our objects first

```
Ls()
```

```
rm(list=ls()) #will remove ALL objects
Ls()
```

#create a list (will be empty first)

```
mylist <- list()
```

#Let’s add something to our list

```
mylist[["foo"]]<-c(1,2,3)
```

#and let’s add some more

```
mylist [["bar"]] <- c("a", "b", "c")
```

#let’s check what we have so far

```
mylist
str(mylist)
```

Data Structure: Lists

```
List of 2
 $ foo: num 1
 $ bar: chr [1:3] "a" "b" "c”
```

#Use the "[[" operator to access one elements in the list
#Use the "[" operator to access several elements
#example: let’s access the element “bar”
mylist[["bar"]]

```
[1] "a" "b" "c”
```

OR
Mylist[[2]]

```
[1] "a" "b" "c”  #This is different from access the second element  in a vector, indeed you are accessing the second element in a list
```

Data Structure: Lists

Why are list particularly important in R? The results of most predefined statistical functions will output results in a list. It is important to access elements in a list to extract those results we need.

Example:

#Let's generate some data and run a linear regression model

```
n<-20
predictor<-rnorm(n)
predicted<-1-2*predictor+rnorm(n)
results<-lm(predicted~predictor)
```

Data Structure: Lists

Results

```
lm(formula = predicted ~ predictor)
Coefficients:
(Intercept)    predictor
      1.062       -1.955
```

summary(results)

```
Call:
lm(formula = predicted ~ predictor)

Residuals:
    Min      1Q  Median      3Q     Max
-1.3395 -0.4880 -0.1855  0.3868  1.8890

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1.0624     0.2126   4.996 9.36e-05 ***
predictor    -1.9548     0.1708 -11.446 1.08e-09 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.9504 on 18 degrees of freedom
Multiple R-squared:  0.8792,    Adjusted R-squared:  0.8725
F-statistic:   131 on 1 and 18 DF,  p-value: 1.076e-09
```

Data Structure: Lists

#Let us take a closer look at this object called “results”
#brace

```
str(results)
```

```
List of 12
 $ coefficients : Named num [1:2] 1.06 -1.95
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "predictor"
 $ residuals    : Named num [1:20] 0.59 1.539 -0.478 0.213 1.869 ...
  ..- attr(*, "names")= chr [1:20] "1" "2" "3" "4" ...
 $ effects      : Named num [1:20] -5.108 -10.878 -0.255 0.465 2.295 ...
  ..- attr(*, "names")= chr [1:20] "(Intercept)" "predictor" "" "" ...
```

etc….

#The “$”represent accessible items in the list

```
str(summary(results)) #reveals another list
```

Data Structure: Lists

```
savemyresults<-summary(results) #save your summary results into a new  object
ls()                            #should be one your objects

#look at your residuals; use the "$" to access the item "residuals
savemyresults$residuals

#OR access the third item in the list
Savemyresults[[3]]

#extract your predictor p-value which is what you would get if you simply typed
"results" in the first place (the significance of your model).

#This is the eighth value in the list item "coefficient"
savemyresults$coefficients
```

```
             Estimate      Std. Error              t value              Pr(>|t|)
(Intercept)  1.062364 (1)    0.2126265  (3)  4.996387 (5)          9.357031e-05 (7)
predictor   -1.954827  (2)   0.1707880  (4) -11.445924 (6)   1.076036e-09 (8)
```

```
savemyresults$coefficients[8]      # This will do the exact same thing:
savemyresults[[4]][8]
[1] 1.076036e-09
```

Delete Elements: Lists

```
#to delete an element in a list
#go back to your first list
      mylist
      str(mylist)
      mylist[["bar"]] <- NULL

str(mylist)
List of 1
$ foo: num 1
```

Data Structure: Attributes

- Will likely be covered in more detail when when writing your functions.
- Briefly….so you can sleep well tonight… attributes are simply meta-data. In the previous example, names of the individual element of a list are an attribute.

Example:

```
 mysecondlist <- list(a=100, b=200, c=300)
str(mysecond list)
List of 3
 $ a: num 1
 $ b: num 2
 $ c: num 3
attributes(mysecondlist)
$names
[1] "a" "b" "c"
```

Data Structure: Attributes

#Attributes are also the names of rows and columns in a data frame.
#Example:

```
dat<-data.frame(x=5:6, y=7:8)
str(dat)
```

```
'data.frame':    2 obs. of  2 variables:
 $ x: int  5 6
 $ y: int  7 8
```

```
attributes(dat)
```

```
$names
[1] "x" "y”
$row.names
[1] 1 2
$class
[1] "data.frame"
```

Data Structure: Attributes

#Let’s go ahead and change the row names of our dataframe
rownames(dat)<-c("participant1","participant2”); names(dat)<-c(“name1”, “name2”)

```
 dat
                name1   name2
participant1        5       7
participant2        6       8
```

attributes(dat)

```
$names
[1] "name1" "name2"

$row.names
[1] "participant1" "participant2"

$class
[1] "data.frame"
```

Data Structure: Attributes

#Attributes are also used to store comments for a function (more about this later in the week). Briefly

#Create a little function that computes the mean which is simply the sum of the value of divided by the number of values:

```
    tt<-function(x){sum(x)/length(x)}
#comment your code (enter one line at the time)

    tt<-function(x)  {
        #Function to compute the mean
        sum(x)/length(x)
            }
str(tt)
```

Data Structure: Recycling Rule

In vector arithmetic, R performs element by element operations

```
#For example try
    x<-c(2,4,6)
    y<-c(7,9,10)
    x+y
    [1]  9 13 16
```

- This is straight-forward because the vectors have the same lengths

- But what happens when vectors are of different length?

Data Structure: Recycling Rule

- In this case R evokes the “Recycling Rule”
- When the shorter vector is exhausted, R returns to the beginning “recycling” its elements but continues taking elements from the longer vector until the operation is complete. It will recycle the shorter-vector elements over and over until necessary

V1	V2
1	1
2	2
3	3
4	
5	
6	

Data Structure: Recycling Rule

- In this case R evokes the “Recycling Rule”
- When the shorter vector is exhausted, R returns to the beginning “recycling” its elements but continues taking elements from the longer vector until the operation is complete. It will recycle the shorter-vector elements over and over until necessary

(1:6)+(1:3)

V1	V2	1:6+1:3
1	1	2
2	2	4
3	3	6
4	1	5
5	2	7
6	3	9

Data Structure: Recycling Rule

The Recycling rule is also called by Functions.
For example:

cbind(1:7)

```
[1,] 1
[2,] 2
[3,] 3
[4,] 4
[5,] 5
[6,] 6
[7,] 7
```

cbind(1:4)

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

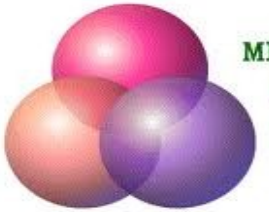
cbind(1:7, 1:4)

```
[,1]  [,2]
[1,] 1   1
[2,] 2   2
[3,] 3   3
[4,] 4   4
[5,] 5   1
[6,] 6   2
[7,] 7   3
```

(1:7) + (1:4)

```
[1]  2  4  6  8  6  8 10
Warning message: #When the shorter vector is not a
multiple of the longer one
In (1:7) + (1:4) :
 longer object length is not a multiple of shorter object
length
```

Acknowledgements:



MRC Social, Genetic & Developmental Psychiatry Centre

Leo Schalkwyk

Organisers behind this year *Introduction to R* summer school including speakers and demonstrators

If you have questions I will be around for most of the week or you can always email me for a slower response

karim.malki@kcl.ac.uk

KING'S College LONDON