Data Analytics and Visualizations in R - Exercises

Contents

1 Basic R Data Structures

1.1 Types

What are the scalar types in R?

```
# Scalars are just vectors of length one
```

1.2 Weirdness of R

What is the major difference between atomic vectors and lists? How can you turn a list into an atomic vector?

In order to check if an object is of a certain type you can use `is.[type](object)` ,e.g. `is.integer(object)`

Can you use the `is.vector()` function to understand whether a data structure is a vector? If not, what are the functions that you can use for this purpose?

```
# Atomic vectors can contain elements of the same type. The elements of a list
# can have different types.

# Yes, we can use the `is.vector()` function to understand is the structure is a
# vector. It will return TRUE or FALSE.

a <- list(c(1,2,3), "bla", TRUE)
print(a)
```

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

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

```
is.vector(a)
```

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

```
# Best response: If possible you should use type specific coercions like
# `as.numeric()` or `as.character()`. But since lists are heterogenous, this
# might not work. A more general function is `unlist()`,  which returns the list
# into a vector of the most general type. Notice this difference:

a <- list(c(1,2,3), "bla", TRUE)
unlist(a)
```

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

```
as.character(a)
```

```
## [1] "c(1, 2, 3)" "bla"        "TRUE"
```

```
# Generally you can, but here comes the weird part: `is.vector()` will only return `TRUE` if the vector
# no attributes `names`. Therefore more specific functions like `is.atomic()` or `is.list()` functions
# be used to test if an object is actually atomic vectoror a list
```

1.3 Atomic Vector Concatenation

What happens when you try to generate an atomic vector with `c()`which is composed of different types of elements? What is the `mean()` of a logical vector?

```
# When we attempt to combine different types they will be coerced to the most
# flexible type. Types from least to most flexible are: logical, integer, double
#  and character.

str(c("a", 1)) # 1 corced to char
```

```
##  chr [1:2] "a" "1"
```

```
# As TRUE is encoded as 1 and FALSE as 0, the mean is the number of TRUEs
# devided by the vector length.
mean(c(TRUE, FALSE,FALSE))
```

```
## [1] 0.3333333
```

1.4 Vector Concatenation

Compare X and Y where X and Y are defined as follows. What is the difference?

```
x <- list(list(1,2), c(3,4))
y <- c(list(1,2), c(3,4))
```

```
# Answer
# X will combine seveal lists into one. Given a combination of atomic vectors
# and lists, y will coerce the vectors to lists before combining them.
str(x)
```

```
## List of 2
##  $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
##  $ : num [1:2] 3 4
```

```
str(y)
```

```
## List of 4
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ : num 4
```

1.5 From Vectors to `data.frames`

First, create three named numeric vectors of size 10, 11 and 12 respectively in the following manner:

- One vector with the "colon" approach: *from:to*
- One vector with the `seq()` function: *seq(from, to)*
- And one vector with the `seq()` function and the by argument: *seq(from, to, by)*

For easier naming you can use the vector `letters` or `LETTERS` which contain the latin alphabet in small and capital, respectively. In order to select specific letters just use e.g. `letters[1:4]` to get the first four letters. Check their types. What is the outcome? Where do you think does the difference come from?

Then combine all three vectors in a list. Check the attributes of the vectors and the list. What is the difference and why?

Finally coerce the list to a `data.frame` with `as.data.frame()`. Why does it fail and how can we fix it? What happend to the names?

Hint: If list elements have no names, we can access them with the double brackets and an index, e.g. `my_list[[1]]`

```
# Answer

# A. create vectors
aa <- 1:10
names(aa) <- letters[aa]
aa
```

```
##  a  b  c  d  e  f  g  h  i  j
##  1  2  3  4  5  6  7  8  9 10
```

```
bb <- seq(1, 11)
names(bb) <- letters[bb]
bb
```

```
##  a  b  c  d  e  f  g  h  i  j  k
##  1  2  3  4  5  6  7  8  9 10 11
```

```
cc <- seq(1, 12, by=1)
names(cc) <- letters[cc]

typeof(aa)
```

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

```r
typeof(bb)
```

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

```r
typeof(cc)
```

```
## [1] "double"
```

```r
# B. Combine all three vectors in a list

my_list <- list(aa, bb, cc)
my_list
```

```
## [[1]]
##  a  b  c  d  e  f  g  h  i  j
##  1  2  3  4  5  6  7  8  9 10
##
## [[2]]
##  a  b  c  d  e  f  g  h  i  j  k
##  1  2  3  4  5  6  7  8  9 10 11
##
## [[3]]
##  a  b  c  d  e  f  g  h  i  j  k  l
##  1  2  3  4  5  6  7  8  9 10 11 12
```

```r
attributes(aa)
```

```
## $names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```r
attributes(bb)
```

```
## $names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k"
```

```r
attributes(cc)
```

```
## $names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

```r
attributes(my_list)
```

```
## NULL
```

```r
# C. Coerce to data.frames

# my_df <- as.data.frame(my_list)# fails

# Fixing the length
my_list[[1]] <- c(my_list[[1]], NA, NA)
my_list[[2]] <- c(my_list[[2]], NA)

my_df <- as.data.frame(my_list)
names(my_df) <- LETTERS[1:3]
my_df
```

```
##    A  B  C
## a  1  1  1
## b  2  2  2
## c  3  3  3
## d  4  4  4
## e  5  5  5
## f  6  6  6
## g  7  7  7
## h  8  8  8
## i  9  9  9
## j 10 10 10
## k NA 11 11
##   NA NA 12
```

1.6 Attributes

Take again our `data.frame` from Question 5.

- Change the row names and the column names of the `data.frame` to capital letters (or small letters, if they are already capital.
- Change the `class` attribute to *list*. What happens?
- Change it now to any name you like. What happens now? What happens if you remove the class attribute

```r
# Answer
# A. One possible way through attributes

attributes(my_df)
```

```
## $names
## [1] "A" "B" "C"
##
## $row.names
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" ""
##
## $class
## [1] "data.frame"
```

```r
attr(my_df, "names") <- letters[1:3]
attr(my_df, "row.names") <- LETTERS[1:12]
my_df
```

```
##    a  b  c
## A  1  1  1
## B  2  2  2
## C  3  3  3
## D  4  4  4
## E  5  5  5
## F  6  6  6
## G  7  7  7
## H  8  8  8
## I  9  9  9
## J 10 10 10
## K NA 11 11
## L NA NA 12
```

```r
# Or through accessor functions

names(my_df) <- LETTERS[1:3]
```

```r
row.names(my_df) <- letters[1:12]
my_df
```

```
##     A  B  C
## a   1  1  1
## b   2  2  2
## c   3  3  3
## d   4  4  4
## e   5  5  5
## f   6  6  6
## g   7  7  7
## h   8  8  8
## i   9  9  9
## j  10 10 10
## k  NA 11 11
## l  NA NA 12
```

```r
# B.

attr(my_df, "class") <- "list"
my_df
```

```
## $A
##  [1]  1  2  3  4  5  6  7  8  9 10 NA NA
##
## $B
##  [1]  1  2  3  4  5  6  7  8  9 10 11 NA
##
## $C
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
##
## attr(,"row.names")
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
## attr(,"class")
## [1] "list"
```

```r
# Answer - the data.frame coerced to a list

# C
attr(my_df, "class") <- "Batman"
my_df
```

```
## $A
##  [1]  1  2  3  4  5  6  7  8  9 10 NA NA
##
## $B
##  [1]  1  2  3  4  5  6  7  8  9 10 11 NA
##
## $C
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
##
## attr(,"row.names")
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
## attr(,"class")
## [1] "Batman"
```

```r
# Answer - Nothing changes
```

1.7 Factors

- What is the difference between a Factor and a Vector?
- Create a vector of length 30 with three levels *Rita Repulsa, Lord Zedd* and *Rito Revolto* and equal length for each level
- What happens if you replace the second element of the vector with *Shredder*

```r
# Answer
# A. A factor is a vector that can contain only predefined values, and is used
# to store categorical data. It is stored as an integer with a character string
# associated with each integer value

# B.

x <- gl(n=3, k=10, length=30, labels=c("Rita Repulsa", "Lord Zedd", "Rito Revolto"))
str(x)
```

```
##  Factor w/ 3 levels "Rita Repulsa",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```r
levels(x)
```

```
## [1] "Rita Repulsa" "Lord Zedd"    "Rito Revolto"
```

```r
attributes(x)
```

```
## $levels
## [1] "Rita Repulsa" "Lord Zedd"    "Rito Revolto"
##
## $class
## [1] "factor"
```

```r
# C
x[2] <- "Shredder"
```

```
## Warning in `[<-.factor`(`*tmp*`, 2, value = "Shredder"): invalid factor
## level, NA generated
```

```r
# It doesn't work. We get the error 'NA generated'
```

1.8 More fun with factors

```r
f1 <- factor(letters)
levels(f1) <- rev(levels(f1))
f2 <- rev(factor(letters))
f3 <- factor(letters, levels = rev(letters))
```

The function `rev` reverses the order of an orderable object. What is the difference between f1, f2 and f3? Why?

```r
# Answer
f1 <- factor(letters)
levels(f1) <- rev(levels(f1))

# f1 goes from a to z and when we apply the levels(f1), z will become 1 and a=26
```

```r
f2 <- rev(factor(letters))

# f2 goes from z to a. but the levels are not changed.

f3 <- factor(letters, levels = rev(letters))

# f3 goes from a - z, but the underlying encoding goes from z = 1 to a = 26.
# We create the vector with the letters a to z BUT the mapped integer structure
# 26 to 1. Hence the levels but not the vector are reversed

f3
```

```
##  [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
## Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a
```

```r
# Reversing f3 will give f1

rev(f3)
```

```
##  [1] z y x w v u t s r q p o n m l k j i h g f e d c b a
## Levels: z y x w v u t s r q p o n m l k j i h g f e d c b a
```

1.9 Creating data.frames

Create a data.frame with 26 rows like this: Only the first and the last six rows are shown. Hint: Instead of the workaround with list you can also use simply `data.frame(column_name = column_vector, ...)`

```r
aa <- seq(1:26)
bb <- seq(from=4, to=4*26, by=4)
cc <- rep(seq(1, 26, 2), each=2)
df <- data.frame(V1 = aa, V2 = bb, V3 = letters[cc])
head(df)
```

```
##   V1 V2 V3
## 1  1  4  a
## 2  2  8  a
## 3  3 12  c
## 4  4 16  c
## 5  5 20  e
## 6  6 24  e
```

```r
tail(df)
```

```
##    V1  V2 V3
## 21 21  84  u
## 22 22  88  u
## 23 23  92  w
## 24 24  96  w
## 25 25 100  y
## 26 26 104  y
```

1.10 Combining `data.frames`

Now take the previous data.frame from Question 10 and reproduce the following `data.frame`. Only the first and the last six rows are shown **Hint:** In order to combine to data.frames by column you can use `cbind(df1, df2, ...)`

help(cbind)

```r
df[,1] <- NULL
dd <- rev(rep(seq(1, 26, 2), each = 2))
ee <- seq(0, 1.6, length.out = 26)
df2 <- data.frame(V4 = dd, V5 = ee)
binded_df <- cbind(df2, df)
head(binded_df)
```

```
##   V4    V5 V2 V3
## 1 25 0.000  4  a
## 2 25 0.064  8  a
## 3 23 0.128 12  c
## 4 23 0.192 16  c
## 5 21 0.256 20  e
## 6 21 0.320 24  e
```

```r
tail(binded_df)
```

```
##    V4    V5  V2 V3
## 21  5 1.280  84  u
## 22  5 1.344  88  u
## 23  3 1.408  92  w
## 24  3 1.472  96  w
## 25  1 1.536 100  y
## 26  1 1.600 104  y
```

1.11 Computation on data.frames

Create the data.frame *df* with `df <- as.data.frame(matrix(runif(9e6), 3e3, 3e3))` This will create a data.frame with 3000 columns and rows and a total of 9mil values.

Now compute the sum of any row, then compute the sum of any column. Measure the time for both operations. Why are the times different, although the size is the same?

- **Hint1:** The time is measured with the function `system.time(my_function_call())`, e.g: `system.time(mean(my_vector))`
- **Hint2:** The sum can be computed with the sum function `sum(my_vector)`
- **Hint2:** Columns and rows are selected by single brackets. Rows: `df[row_number,]`, Columns: `df[,col_number]`

```r
# Answer

df <- as.data.frame(matrix(runif(9e6), 3e3, 3e3))

# rows
system.time(res <- sum(df[1,]))
```

```
##    user  system elapsed
##   0.024   0.000   0.025
```

```r
res
```

```
## [1] 1507.325
```

```r
# columngs
system.time(res2 <- sum(df[,1]))
```

```
##    user  system elapsed
##       0       0       0
```

```r
res2
```

```
## [1] 1475.916
```

```r
# Look at the structure of the objects over which we are computing the sum
# Column
str(df[1,])
```

```
## 'data.frame':    1 obs. of  3000 variables:
##  $ V1   : num 0.564
##  $ V2   : num 0.624
##  $ V3   : num 0.295
##  $ V4   : num 0.881
##  $ V5   : num 0.974
##  $ V6   : num 0.916
##  $ V7   : num 0.689
##  $ V8   : num 0.383
##  $ V9   : num 0.377
##  $ V10  : num 0.751
##  $ V11  : num 0.703
##  $ V12  : num 0.07
##  $ V13  : num 0.727
##  $ V14  : num 0.316
##  $ V15  : num 0.856
##  $ V16  : num 0.259
##  $ V17  : num 0.819
##  $ V18  : num 0.019
##  $ V19  : num 0.179
##  $ V20  : num 0.135
##  $ V21  : num 0.473
##  $ V22  : num 0.946
##  $ V23  : num 0.342
##  $ V24  : num 0.647
##  $ V25  : num 0.836
##  $ V26  : num 0.844
##  $ V27  : num 0.827
##  $ V28  : num 0.962
##  $ V29  : num 0.64
##  $ V30  : num 0.563
##  $ V31  : num 0.466
##  $ V32  : num 0.933
##  $ V33  : num 0.783
##  $ V34  : num 0.277
##  $ V35  : num 0.47
##  $ V36  : num 0.657
##  $ V37  : num 0.364
##  $ V38  : num 0.966
##  $ V39  : num 0.546
##  $ V40  : num 0.445
##  $ V41  : num 0.771
##  $ V42  : num 0.429
##  $ V43  : num 0.53
##  $ V44  : num 0.569
##  $ V45  : num 0.381
##  $ V46  : num 0.469
##  $ V47  : num 0.85
##  $ V48  : num 0.736
##  $ V49  : num 0.912
##  $ V50  : num 0.975
##  $ V51  : num 0.0319
##  $ V52  : num 0.705
##  $ V53  : num 0.654
##  $ V54  : num 0.658
##  $ V55  : num 0.468
##  $ V56  : num 0.508
##  $ V57  : num 0.501
##  $ V58  : num 0.306
##  $ V59  : num 0.737
##  $ V60  : num 0.564
##  $ V61  : num 0.278
##  $ V62  : num 0.632
##  $ V63  : num 0.297
##  $ V64  : num 0.0557
##  $ V65  : num 0.592
##  $ V66  : num 0.644
##  $ V67  : num 0.412
##  $ V68  : num 0.234
##  $ V69  : num 0.695
##  $ V70  : num 0.104
##  $ V71  : num 0.471
##  $ V72  : num 0.549
##  $ V73  : num 0.0884
##  $ V74  : num 0.196
##  $ V75  : num 0.817
##  $ V76  : num 0.985
##  $ V77  : num 0.00622
##  $ V78  : num 0.672
##  $ V79  : num 0.766
##  $ V80  : num 0.129
##  $ V81  : num 0.678
##  $ V82  : num 0.925
##  $ V83  : num 0.847
##  $ V84  : num 0.825
##  $ V85  : num 0.287
##  $ V86  : num 0.181
##  $ V87  : num 0.615
##  $ V88  : num 0.801
##  $ V89  : num 0.94
##  $ V90  : num 0.114
##  $ V91  : num 0.273
##  $ V92  : num 0.0643
##  $ V93  : num 0.989
##  $ V94  : num 0.463
##  $ V95  : num 0.914
##  $ V96  : num 0.675
##  $ V97  : num 0.196
##  $ V98  : num 0.0357
##  $ V99  : num 0.866
##   [list output truncated]
```

```r
# Row
str(df[,1])
```

```
##  num [1:3000] 0.564 0.138 0.613 0.997 0.975 ...
```

```r
# As we can see the extracted column is a numeric vector. But the extracted
# row is a list. Under the hood the sum function is iterating in C/Fortran
# over the specific structure. Iterating over a native array of doubles is
# faster, than iterating over a structure, where at each position, the value
# has to be retrieved from an object possibly strored somewhere further away
# in memory.
```

1.12 Missing Values

- If NA is just a placeholder for a missing value of the same type and Infinity is of type double, why is Infinity plus NA not Infinity?

Hint:

```r
paste(paste(rep((Inf - Inf), 20), collapse = ""), "Batman!")
```

```
## [1] "NaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaN Batman!"
```

```r
# Answer:
# Because infinity is not a well defined number, but a concept with
# the type 'double' in R. Adding or subtracting any number from infinity
# will give infinity. However NA is placeholder for any value of the same
# type and therefore also for infinity. As infinity plus/minus infinity
# is not defined, adding NA to infinity can theoretically lead to
# Nan (not a number). Therefore Inf + NA will produce NA.

Inf + NA
```

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

```r
Inf + 1
```

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

```r
Inf - Inf
```

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