# Week 1. Big Data Analytics - The R Language

Hyeonsu B. Kang
hyk149@eng.ucsd.edu

March 2016

## 1 Data Representations in R

### 1.1 Vectors, Matrices, Factors, Lists, Data Frames

Exercise 1. **Vectors** I. Create the following vectors:

(1) An empty vector

(2) $(1, 2, 3, \cdots, 99, 100)$

(3) $(100, 99, 98, \cdots, 2, 1)$

(4) $(1, 2, 3, \cdots, 99, 100, 99, 98, \cdots, 2, 1)$

(5) $(1.0, 1.5, 2.0, \cdots, 10.0)$

(6) The function `rep` (please look up the help) would be helpful for this and the next question:

$$(2, 3, 5, 7, 2, 3, 5, 7, \cdots, 2, 3, 5, 7),$$

where there are 50 occurrences of 2.

(7) Create a vector

$$(2, 2, 2, 3, 3, 3, 5, 5, 5, 7, 7, 7, \cdots, 2, 2, 2, 3, 3, 3, 5, 5, 5, 7, 7, 7),$$

where there are 30 occurrences of 2.

(8) The function `paste` (please look up the help) would be helpful for this question: `("Item1", "Item2", ···, "Item50")`.

Exercise 2. **Vectors** II. Vectors can only contain one atomic type. If you try to combine different types, R will create a vector that is the least common denominator: the type that is easiest to coerce to.

(1) Guess what the following statements do without running them first:

```
xx <- c(1.7, "a")
xx <- c(TRUE, 2)
xx <- c("a", TRUE)
```

(This is called **implicit coercion**). The coercion rule goes `logical -> integer -> numeric -> complex -> character`. You can also coerce vectors explicitly using `as.<class name>`.

(2) Guess what the following statements would do without running them first:

```
x <- c(10, 12, 45, 33, 57)
as.character(x)
as.complex(x)
x <- 0:6
as.logical(x)
```

(3) Execute the following statements:

```
x <- 0:10
tail(x, n=2)
head(x, n=2)
length(x)
```

(4) Execute the following statements:

```
x <- 1:4
names(x) <- c("a", "b", "c", "d")
x
```

Exercise 3. **Matrices**. Matrices are really just atomic vectors, with added dimension attributes.

(1) We can create one with the `matrix` function. Let's generate some random data:

```
set.seed(1) # make sure the system creates reproducible random numbers
x <- matrix(rnorm(18), ncol=6, nrow=3)
x
str(x)
```

(2) What do you think will be the result of `length(x)`?

(3) Create another matrix, this time containing the numbers $1:50$, with 5 columns and 10 rows. Did the `matrix` function fill your matrix by column, or by row, as its default behavior? See if you can figure out how to change this. (hint: read the documentation for `matrix`)

Exercise 4. **Factors**. Factors are special vectors that represent categorical data. Factors can be ordered or unordered. They are important when modelling functions and in plot methods.

(1) Factors can only contain predefined values, and we can create one with the `factor` function:

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

(2) This reveals something important: while factors look (and often behave) like character vectors, they are actually integers under the hood, and here, we can see that "no" is represented by a 1, and "yes" a 2. By default, R always sorts levels in alphabetical order. You can change this by specifying the levels:

```
# ‘‘case’’ will be assigned with 1 after the following statement
x <- factor(c("case", "control", "control", "case"), levels = c("control", "case"))
str(x)
```

(3) Sometimes, the order of the factors does not matter, other times you might want to specify the order because it is meaningful (e.g., "low", "moderate", "considerable", "high", "extreme") or it is required by particular type of analysis. Additionally, specifying the order of the levels allows us to compare levels:

```
scale <- factor(c("high", "moderate", "high", "low", "extreme"), [cont'd]
levels=c("low", "moderate", "considerable", "high", "extreme"), ordered=TRUE)
levels(scale)
```

Exercise 5. **Lists**. If you want to combine different types of data, you will need to use lists. Lists act as containers, and can contain any type of data structure, even themselves!

(1) Lists can be created using `list` or coerced from other objects using `as.list()`:

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

(2) Lists can contain more complex objects:

```
xlst <- list(a = "Research Bazaar", b = 1:10, data = head(iris))
xlst
```

(3) In the above case `xlst` contains a character vector of length one, a numeric vector with 10 entries, and a small data frame from one of R's many preloaded datasets (see `?data`). We have also given each list element a name, which is why you see `$a` instead of `[[1]]`. List can also contain themselves:

```
list(list(list(list())))
```

(4) Let's incorporate one of the data sets already in R. Using `data()` will show the list of the data sets. Create a list that contains both Biochemical Oxygen Demand data, Carbon Dioxide Uptake in Grass Plants data.

Exercise 6. **Data frames**. Data frames are similar to matrices, except each column can be a different atomic type. A data frames is the standard structure for storing and manipulating rectangular data sets. Underneath the hood, data frames are really lists, where each element is an atomic vector, with the added restriction that they are all the same length. As you will see, if we pull out one column of a data frame, we will have a vector.

(1) **Creating data frames** - Data frames can be created manually with the `data.frame` function:

```
df <- data.frame(id = c('a', 'b', 'c', 'd', 'e', 'f'), x = 1:6, y = c(214:219))
df
```

Try using the `length` function to query your data frame `df`. Does it give the result you expect?

(2) **Adding columns to a data frame** - Execute the following statements to add a column to `df`:

```
df <- cbind(df, 6:1)
df
```

Note that R automatically names the column. We may decide to change the name by assigning a value using the `names` function or by providing a name when we add the column:

```
df <- cbind(df, 6:1)
names(df)[4] <- 'SixToOne'
df
df <- cbind(df, caps=LETTERS[1:6])
df
```

(3) **Adding rows to a data frame** - To add a row we use `rbind`:

```
df <- rbind(df, list("g", 11, 42, 0, "G"))
```

Note that we add the row as a list, because we have multiple types across the columns. Nevertheless, this doesn't work as expected! What do the error messages tell us? It appears that R was trying to append "g" and "G" as factor levels. Why? Let's examine the first column. We can access a column in a `data.frame` by using the $ operator:

```
class(df$id)
str(df)
```

When we created the data frame, R automatically made the first and last columns into factors, not character vectors. There are no pre-existing levels "g" and "G", so the attempt to add these values fails. A row was added to the data frame, only there are missing values in the factor columns:

```
df
```

There are two ways we can work around this issue: (1) We can convert the factor columns into characters. This is convenient, but we lose the factor structure. (2) We can add factor levels to accommodate the new additions. If we really do want the columns to be factors, this is the correct way to proceed. We will illustrate both solutions in the same data frame, using the first and last columns.

```
df$id <- as.character(df$id) # convert to character
class(df$id)
levels(df$caps) <- c(levels(df$caps), 'G') # add a factor level
class(df$caps)
```

Now let's try adding that row again.

```
df <- rbind(df, list("g", 11, 42, 0, 'G'))
tail(df, n=3)
```

(4) **Deleting rows and handling** `NA` - We successfully added the last row, but we have an undesired row with two `NA` values. How do delete it? The following ways perform identical deletion of a row containing `NA`

```
df[-7, ] # The minus sign tells R to delete the row
df[complete.cases(df), ] # Return 'TRUE' when no missing values
na.omit(df) # Another function for the same purpose
df <- na.omit(df)
```

(5) **Combining data frames** - We can also row-bind data frames together, but notice what happens to the row names:

```
rbind(df, df)
```

R is making sure that row names are unique. You can restore sequential numbering by setting row names to NULL:

```
df2 <- rbind(df, df)
rownames(df2) <- NULL
df2
```

(6) Create a dataframe `df` that contains the `rock` dataset.
(7) Add a column that has numbers from 1 to 48.
(8) Name the added column to "SampleNum".
(9) Add a row that has values 9999, 1500.0, 0.315, 900.0 and 49.
(10) View the last 7 rows.
(11) Remove the row 23 to 27 and 49.

## 1.2 Subsetting Data

R has many powerful subset operators and mastering them will allow you to easily perform complex operations on any kind of dataset. There are six different ways we can subset any kind of object, and three different subsetting operators for the different data structures.

Exercise 1. **Accessing elements using their indices**. Create the following vector:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
x
```

To extract elements of a vector, we can give their corresponding index, starting from one:

```
x[1]
```

or we can ask for multiple elements at once:

```
x[c(1, 3)]
```

or even ask for the same element multiple times:

```
x[c(1, 1, 3)]
```

or slices of the vector:

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

In the above, the `:` operator just creates a sequence of numbers from the left element to the right. I.e. `x[1:4]` is equivalent to `x[c(1, 2, 3, 4)]`.

If we ask for a number outside of the vector, R will return missing values. Execute the following and look at the result:

```
x[6]
```

The displayed result – `<NA> NA` – indicates a vector of length one containing an `NA`, whose name is also `NA`.

If we ask for the $0^{th}$ element, we get an empty vector. Execute the following and look at the result:

```
x[0]
```

Exercise 2. **Skipping and removing elements**. If we use a negative number as the index of a vector, R will return every element *except* for the one specified. Execute the following and look at the result:

```
x[-2]
```

Or we can skip multiple elements. Execute the following and look at the result:

```
x[c(-1, -5)] # or equivalently, x[-c(1, 5)]
```

To remove elements from a vector, we need to assign the results back into the variable:

```
x <- x[-4]]
x
```

Given the following code:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x),
```

Come up with at least 3 different commands that will produce the following output:

```
  b   c   d
6.2 7.1 4.8
```

Compare notes with your neighbour. Did you have different strategies?

Or we can extract elements by using their names, instead of indices. Execute the following and look at the result:

```
x[c("a", "c")]
```

To skip (or remove) a single element with a name "a":

```
x[-which(names(x) == "a")]
```

The `which` function returns the indices of all `TRUE` elements of its argument. Remember that expressions evaluate before being passed to functions. Let's break this down so that it's clearer what's happening. Firstly, execute the following and look at the result:

```
names(x) == "a"
```

`which` then converts this to an index. Execute the following and look at the result:

```
which(names(x) == "a")
```

Finally, the negation sign lets R to skip the corresponding element.

Skipping multiple elements by their names is similar, but uses a different comparison operator:

```
x[-which(names(x) %in% c("a", "c"))] # Look up by help("%in%") or ?"%in%"
```

Exercise 3. **Subsetting through other logical operations**. We can also more simply subset through logical operations. Execute the followings and look at the result:

```
x <- 1:3
names(x) <- c('a', 'a', 'a')
x[c(TRUE, TRUE, FALSE, FALSE)]
x[c(TRUE, FALSE)]
x[x > 7]
```

In case of `x[c(TRUE, FALSE)]`, the logical vector is re-cycled to the length of the vector we're subsetting.

Given the following code, write a subsetting command to return the values in x that are greater than 4 and less than 7:

```
x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
names(x) <- c('a', 'b', 'c', 'd', 'e')
print(x)
```

Exercise 4. **Handling special values**

There are a number of special functions you can use to filter out missing, infinite, or undefined data: `is.na` will return all positions in a vector, matrix, or data.frame containing `NA`. Likewise, `is.nan` and `is.infinited` will do the same for `NaN` and `Inf`.

`is.finite` will return all positions in a vector, matrix, or data.frame that do not contain `NA`, `NaN` or `Inf`.

`na.omit` will filter out all missing values from a vector.

(1) **Factor subsetting**. Factor subsetting works the same way as vector subsetting. Execute the following and look at the result:

```
f <- factor(c("a", "a", "b", "c", "c", "d"))
f[f == "a"]
f[f %in% c("b", "c")]
f[1:3]
f[-3]
```

`NOTE`: skipping elements will not remove the level even if no more of that category exists in the factor.

(2) **Matrix subsetting**. In case of matrix subsetting, it takes two arguments: the first applying to the rows, the second to its columns. Execute the following and look at the result:

```
set.seed(1)
m <- matrix(rnorm(6*4), ncol=4, nrow=6)
m[3:4, c(3,1)]
```

You can leave the first or second arguments blank to retrieve all the rows or columns respectively:

```
m[, c(3,4)]
m[3,]
```

If you want to keep the output as a matrix, you need to specify a third argument:

```
m[3, , drop=FALSE]
```

`NOTE`: Matrices are laid out in **column-major** format by default. If you wish to populat the matrix by row, specify `byrow=TRUE`:

```
matrix(1:6, nrow=2, ncol=3, byrow=TRUE)
```

`NOTE`: Compare this with `matrix(1:6, nrow=2, ncol=3)`.

(3) **List subsetting**. There are three functions used to subset lists: `[]`, `[[]]` and `$`. Using `[]` will always return a list. We can subset elements of a list exactly the same was as atomic vectors using `[]`. Execute the following and look at the result:

```
xlst <- list(a = "Software Carpentry", b = 1:10, data = head(iris))
xlst[1:2]
```

To extract individual elements of a list, you need to use the double-square bracket function:

```
xlst[[1]]
```

The `$` function is a shorthand way for extracting elements by name:

```
xlst$data
```

Given a linear model:

```
mod <- aov(pop   lifeExp, data=gapminder),
```

extract the residual degrees of freedom (hint: `attributes() will help you`)

(4) **Data frames subsetting**. The data frames are really just lists underneath the hood, so similar rules apply. However, they are also 2-dimensional objects. `[]` with one argument will act the same was as for lists, where each list element corresponds to a column. The resulting object will be a data frame. Similarly, `[[]]` will act to extract a single column. And `$` provides a convenient shorthand to extract columns by name. Execute the following and look at the result:

[1] Create a factor `f` that has 9 elements (3 "a"'s, 1 "b", 1 "c", 1 "d", 2 "e"'s and 1 "f") and default levels.

[2] Create a vector `v` that has 9 elements ("a", "b", "a", "c", "d", "e", "e", "a", "f") with names 1 to 9.

[3] Extract elements with levels "b" and "c" in f.

[4] Extract elements with names "1" and "2" in v.

[5] Delete the second "a" from the left in `f`.

[6] Delete the "1" and "5" name elements in `v`.

[7] Create a matrix `m` with the following code:

```
set.seed(1)
m <- matrix(rnorm(6*5), nrow=6, ncol=5))
```

[8] Subset `m` for row 2, 3 and column 4, 5.

[9] Extract the (2, 3) element of `m`.

[10] Extract the third row of `m` as a matrix.

[11] Create a data frame `df` with a dataset `pressure`.

[12] Subset `df` with "pressure".

[13] Extract the `temperature` column data.

# 2 Practice Questions

## 2.1 Problem 1. A complex list

Create a list that contains (in the following order)

a list that has numbers $1, 2, \cdots, 99$ in it,
a vector that has all of the uppercase alphabets A, B, $\cdots$,
a $3 \times 3$ matrix that contains number 1 to 9 row-filled
and a 2-level factor with a "high" and a "low" in default ordering

## 2.2 Problem 2. A symmetrical matrix

Create a square matrix that has 81 elements of `1, 2, 3` and that is column-filled.

$$\begin{matrix} 1 & 1 & \cdots & 1 \\ 2 & 2 & \cdots & 2 \\ 3 & 3 & \cdots & 3 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \\ 2 & 2 & \cdots & 2 \\ 3 & 3 & \cdots & 3 \end{matrix}$$

## 2.3 Problem 3. Subsetting a list

Given the following list:

```
xlist <- list(a = "Software Carpentry", b = 1:10, data = head(iris))
```

extract the number 2 from xlist.