Lab 1 Basic R

Creating objects in R

You can get output from R simply by typing math in the console:

```
5 + 5
## [1] 10
12 / 6
## [1] 2
```

In order to do more complex things we need to assign values to variables:

```
X <- 12
```

Variable is an object that can contain some value like number, string or even table.

In order to assign value to variable you can use next operators:

= or <-

There is no sufficient difference between then and during this course you can use both of them whenever you want. (Tip: to put <- just press ALT+- in RStudio).

Variable name cannot start with number or contain spacebars(“ “) and also R is case sensitive language. This means next variables var, Var and vAr are different and can contain different values.

Also some variable names are reserved by R (e.g., if, else, for, see here for a complete list).

Let’s look at example

```
varX <- 5 // we assign value to variable varX
varX * 10 // multiply it by 10
## [1] 50 // automatically obtaining result
```

Now we can assign new value to varX

```
varX <- 6
## [1] 6
varX * 10 // multiply it by 10
## [1] 6
## [1] 60
```

Comments

In order to leave some notes in the code, you can use comments. Simply go to the end of the line where you want leave a comment and put # or // and write youre note on the right side of it.

Functions and their arguments

Functions can be used to execute the same calculations on different set of values.

For example:

```
b <- sqrt(a)
```

Here *sqrt()* is a function that takes value from *a* variable, calculates square root from it and assigns result to *b.*

Let's try a function that can take multiple arguments: `round()`.

Note: use ?round to see help on this function.

```
round(3.14159) // takes 1 argument
```

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

By default function round() takes the value and rounds to the nearest whole number.

Now we will use args command and look, which arguments can be accepted by round.

```
args(round)
```

```
## function (x, digits = 0)
```

Now we can use second argument to leave 2 digits after dot:

```
round(3.14159, digits = 2)
```

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

Or if you provide the arguments in the exact same order as they are defined you don't have to name them:

```
round(3.14159, 2)
```

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

And if you do name the arguments, you can switch their order:

```
round(digits = 2, x = 3.14159)
```

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

Vectors and data types

As we mentioned earlier, variable contains some value, but there can be several types of this value.

The basics types of the value in R are:

- Numeric (1,2,3,-4363, 2.2222)
- Character (or String) ("A", "AAAAAAA")

- Logic (TRUE, FALSE)

Apart from basic data types, R also has more complex data types. One of them is vector.

A vector is the most common and basic data type in R that can contain several values.

For example the following is a numeric vector:

```
X <- c(1,5,4,9,0)
X
## [1] 1 5 4 9 0
```

A vector can also contain characters:

```
queue <- c("first", "second", "third")
queue
## [1] "first" "second"   "third"
```

Note that quotes around “first”, “second”, etc. are essential here. Without the quotes R will assume there are variables called `first`, `second` and `third`. As these variables don’t exist in R’s memory, there will be an error message.

There are many functions that allow you to inspect the content of a vector. `length()` tells you how many elements are in a particular vector:

```
length(X)
## [1] 5
length(queue)
## [1] 3
```

Also the vectors can have missing values (NA):

```
z <- c(NA, 3, 14, NA, 33, 17, NA, 41)
```

NA is name reserved by R and can also be used in Character or Logic vector.

Now let us do a few exercises with vectors.

Take previous numeric vector and multiply it:

1. by 2

```
k <- z*2
```

2. by another vector: 1, 0, 0, 2, 5, 10, 0, -1

```
m <- c(1, 0, 0, 2, 5, 10, 0, -1)
n <- z*m
```

Bind 2 vectors:

1, 3, 5, 7, 11, 13, 17, -1

1, 0, 0, 2, 5, 10, 0, -1

1. using cbind

```
a <- c(1, 3, 5, 7, 11, 13, 17, -1)
b <- c(1, 0, 0, 2, 5, 10, 0, -1)
d <- cbind(a,b)
```

2. using rbind

```
e <- rbind(a,b)
```

Check what next functions do:

dim()

is.numeric()

As you have noticed, **all of the elements are the same type of data in vector**. The function `class()` indicates the class (the type of element) of an object:

```
class(x)
## [1] "numeric"
class(queue)
## [1] "character"
```

You can use the `c()` function to add other elements to your vector:

```
Weight_g <- c(50,49,70,45)
weight_g <- c(weight_g, 90) # add to the end of the vector
weight_g <- c(30, weight_g) # add to the beginning of the vector
weight_g
## [1] 30 50 60 65 82 90
```

In the first line, we take the original vector `weight_g`, add the value `90` to the end of it, and save the result back into `weight_g`. Then we add the value `30` to the beginning, again saving the result back into `weight_g`.

We can do this over and over again to grow a vector, or assemble a dataset. As we program, this may be useful to add results that we are collecting or calculating.

Subsetting vectors

If we want to extract one or several values from a vector, we must provide one or several indices in square brackets. For instance:

```
animals <- c("mouse", "rat", "dog", "cat")
```

```
animals[2]
```

```
## [1] "rat"
```

```
animals[c(3, 2)]
```

```
## [1] "dog" "rat"
```

We can also repeat the indices to create an object with more elements than the original one:

```
more_animals <- animals[c(1, 2, 3, 2, 1, 4)]
more_animals
```

```
## [1] "mouse" "rat"   "dog"   "rat"   "mouse" "cat"
```

R indices start at 1. Programming languages like Fortran, MATLAB, Julia, and R start counting at 1, because that's what human beings typically do. Languages in the C family (including C++, Java, Perl, and Python) count from 0 because that's simpler for computers to do.

Conditional subsetting

Another common way of subsetting is by using a logical vector. `TRUE` will select the element with the same index, while `FALSE` will not:

```
weight_g <- c(21, 34, 39, 54, 55)
weight_g[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

```
## [1] 21 39 54
```

Typically, these logical vectors are not typed by hand, but are the output of other functions or logical tests. For instance, if you wanted to select only the values above 50:

```
weight_g > 50    # will return logicals with TRUE for the indices that meet t
he condition
```

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

```
## so we can use this to select only the values above 50
weight_g[weight_g > 50]
```

```
## [1] 54 55
```

You can combine multiple tests using `&` (both conditions are true, AND) or `|` (at least one of the conditions is true, OR):

```
weight_g[weight_g < 30 | weight_g > 50]
```

```
## [1] 21 54 55
```

```
weight_g[weight_g >= 30 & weight_g == 21]
```

```
## numeric(0)
```

The result `numeric(0)` stands for an empty vector with the type `numeric` for its entries. The type starts to matter once you do some operations with this vector.

A common task is to search for certain strings in a vector. One could use the "or" operator `|` to test for equality to multiple values, but this can quickly become tedious. The function `%in%` allows you to test if any of the elements of a search vector are found:

```
animals <- c("mouse", "rat", "dog", "cat")
animals[animals == "cat" | animals == "rat"] # returns both rat and cat
## [1] "rat" "cat"
animals %in% c("rat", "cat", "dog", "duck", "goat")
## [1] FALSE  TRUE  TRUE  TRUE
animals[animals %in% c("rat", "cat", "dog", "duck", "goat")]
## [1] "rat" "dog" "cat"
```

Data frames

Data frames are the *de facto* data structure for most tabular data, and what we use for statistics and plotting.

This is how you create a new data frame manually, using `data.frame` function

```
df <- data.frame(
Name = c("Jhon", "Joseph", "Martin", "Ivan", "Andrea"),
Goods = c("Bread", "Milk", "Apples", "Meat", "Eggs"),
Sales = c(15, 18, 21, NA, 60),
Price = c(34, 52, 33, 44, NA),
stringsAsFactors = FALSE)
```

A data frame can be created by hand, but most commonly they are generated by the functions `read.csv()` or `read.table()`; in other words, when importing spreadsheets from your hard drive (or the web).

A data frame is the representation of data in the format of a table where the columns are vectors that all have the same length. Because the column are vectors, they all contain the same type of data (e.g., characters, integers, factors).

Inspecting `data.frame` Objects

We already saw how the function `str()` can be useful to check the content and the structure of a data frame. Here is a non-exhaustive list of functions to get a sense of the content/structure of the data. Let's try them out!

- Size:
  - `dim(surveys)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the **dim**ensions of the object)
  - `nrow(surveys)` - returns the number of rows
  - `ncol(surveys)` - returns the number of columns
- Content:
  - `head(surveys)` - shows the first 6 rows
  - `tail(surveys)` - shows the last 6 rows
- Names:

    - `names(surveys)` - returns the column names (synonym of `colnames()` for `data.frame` objects)
    - `rownames(surveys)` - returns the row names
- Summary:
    - `summary(surveys)` - summary statistics for each column

Note: most of these functions are "generic", they can be used on other types of objects besides `data.frame`.

Indexing and subsetting data frames

Now imagine that we need specific value from the variable. We can extract it by using [ ] and adding indexes into it. When it comes to vector, we can use:

```
vector[6]
```

to get 6th value from it. However it is different in data.frame, as it has 2 dimensions(rows and columns). In that case we can use 2 indexes instead of one. Row index come first, followed by column index. However, note that different ways of specifying these coordinates lead to results with different classes.

```
df[1, 1]     # first element in the first column of the data frame (as a vecto
r)
df[1, 3]     # first element in the 3rd column (as a vector)
df[, 1]      # first column in the data frame (as a vector)
df[1]        # first column in the data frame (as a data.frame)
df[1:3, 2]   # first three elements in the 2nd column (as a vector)
df[1:3,1:2] # first three elements in the first two columns (as a data.frame)
df[3, ]      # the 3rd element for all columns (as a data.frame)
```

`:` is a special function that creates numeric vectors of integers in increasing or decreasing order, test `1:10` and `10:1` for instance.

You can also exclude certain parts of a data frame using the "`-`" sign:

```
df[,-1]           # The whole data frame, except the first column
```

As well as using numeric values to subset a `data.frame` (or `matrix`), columns can be called by name, using one of the four following notations:

```
df["Names"]        # Result is a data.frame
df[, "Names"]      # Result is a vector
df[["Names"]]      # Result is a vector
df$Names           # Result is a vector
```

For our purposes, the last three notations are equivalent. RStudio knows about the columns in your data frame, so you can take advantage of the autocompletion feature to get the full and correct column name.

$ is specific operator, that can extract subobject from the collection(like dataframe).

Let us do a few exercises:

```
df <- data.frame(
Name = c("Jhon", "Joseph", "Martin", "Ivan", "Andrea"),
Goods = c("Bread", "Milk", "Apples", "Meat", "Eggs"),
Sales = c(15, 18, 21, NA, 60),
Price = c(34, 52, 33, 44, NA),
stringsAsFactors = FALSE)
```

1. Print "head" and "tail" of dataset.

```
head(df)
tail(df)
```

2. Add some rows to dataset. Print again.

```
newRow <- data.frame("Mike", "Oranges", 22, 35)
colnames(newRow) <- colnames(df)
rbind(df, newRow)
```

3. Remove 2nd row of dataset.

```
df <- df[-2,]
```

1. Make function that looks through data.frame and returns Name of a person, who sold the biggest amount of goods.

```
getBiggestName <- function(data){
        max <- which.max(data$Sales)
        return data$Name[max]
}
```

2. Make function that looks through data.frame and calculates how much each seller earned.

```
getProfit <- function(data){
        return df$Sales * df$Price
}
```

Factors

Factors are very useful and are actually something that make R particularly well suited to working with data, so we're going to spend a little time introducing them.

Factors are used to represent categorical data. Factors can be ordered or unordered, and understanding them is necessary for statistical analysis and for plotting.

Factors are stored as integers, and have labels (text) associated with these unique integers. While factors look (and often behave) like character vectors, they are actually integers under the hood, and you need to be careful when treating them like strings.

Once created, factors can only contain a pre-defined set of values, known as *levels*. By default, R always sorts *levels* in alphabetical order. For instance, if you have a factor with 2 levels:

```
gender <- factor(c("male", "female", "female", "male"))
```

R will assign `1` to the level `"female"` and `2` to the level `"male"` (because `f` comes before `m`, even though the first element in this vector is `"male"`). You can check this by using the function `levels()`, and check the number of levels using `nlevels()`:

```
levels(gender)
## [1] "female" "male"
nlevels(gender)
## [1] 2
```

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”, “medium”, “high”), it improves your visualization, or it is required by a particular type of analysis. Here, one way to reorder our levels in the `gender`vector would be:

```
gender # current order
## [1] male   female female male
## Levels: female male
gender <- factor(gender, levels = c("male", "female"))
gender # after re-ordering
## [1] male   female female male
## Levels: male female
```

In R’s memory, these factors are represented by integers (1, 2, 3), but are more informative than integers because factors are self-describing: `"female"`, `"male"` is more descriptive than `1`, `2`. Which one is “male”? You wouldn’t be able to tell just from the integer data. Factors, on the other hand, have this information built in. It is particularly helpful when there are many levels (like the species names in our example dataset).

Converting factors

If you need to convert a factor to a character vector, you use `as.character(x)`.

```
as.character(gender)
## [1] "male"   "female" "female" "male"
```

Converting factors where the levels appear as numbers (such as concentration levels, or years) to a numeric vector is a little trickier. One method is to convert factors to characters and then numbers. Another method is to use the `levels()` function. Compare:

```
f <- factor(c(1990, 1983, 1977, 1998, 1990))
as.numeric(f)               # wrong! and there is no warning...
```

```
## [1] 3 2 1 4 3
as.numeric(as.character(f)) # works...
## [1] 1990 1983 1977 1998 1990
as.numeric(levels(f))[f]     # The recommended way.
## [1] 1990 1983 1977 1998 1990
```

Notice that in the `levels()` approach, three important steps occur:

- We obtain all the factor levels using `levels(f)`
- We convert these levels to numeric values using `as.numeric(levels(f))`
- We then access these numeric values using the underlying integers of the vector `f` inside the square brackets

Lets do some exercises:

1) Take previous data.frame and add column.

2) Generate factor using sample function (n - amount of rows in dataframe):

```
Answer:
ftrs <- sample(as.factor(c("low", "medium", "high")), n,3);
```

3) Add column to data.frame from previouse task.

```
Answer:
cbind(df, ftrs)
```

4) Rename new column to "Efficiency".

```
Answer: colnames(df)[5] <- "Efficiency”
```

5) Sort data.frame by worker's Efficiency.

```
Answer: df <- df[order(df$Efficiency),]
```

Pipelines %>% operator

You can make your code more readable by using the dyplr's *pipe* operator (**%>%**). This operator takes the object from the left and gives it as the first argument to the function on the right. For example the function **f(x, y)** can be written as **x %>% f(y)**.

Let us count the square of x with using pipeline: x <- 100

```
x %>% sqrt()
```

Now let us count the sum of the products where the price is less than 40 and the sale is equal to 15 in the data frame:

```r
df <- data.frame(

Name = c("Jhon", "Joseph", "Martin", "Ivan", "Andrea"),

Goods = c("Bread", "Milk", "Apples", "Meat", "Eggs"),

Sales = c(15, 18, 21, NA, 60),

Price = c(34, 52, 33, 44, NA),

stringsAsFactors = FALSE)

df <- filter(df, Sales == 15 & Price  < 40)
       summarise(df,sum_ Price = sum(Price))

# and if we want it %>% then

df %>%
       filter(Sales == 15 & Price  < 40) %>%
       summarise(sum_ Price = sum(Price))
```