Introduction to R

1. Intro to basics

Arithmetic with R

In its most basic form, R can be used as a simple calculator. Consider the following arithmetic operators:

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Exponentiation: ^
- Modulo: %%

The ^ operator raises the number to its left to the power of the number to its right: for example 3^2 is 9.

The modulo returns the remainder of the division of the number to the left by the number on its right, for example 5 modulo 3 or 5 %% 3 is 2.

Basic data types in R

R works with numerous data types. Some of the most basic types to get started are:

- Decimal values like 4.5 are called numerics.
- Natural numbers like 4 are called integers. Integers are also numerics.
- Boolean values (TRUE or FALSE) are called logical.
- Text (or string) values are called characters.

Note how the quotation marks on the right indicate that "some text" is a character.

What's that data type?

Check the data type of a variable beforehand. You can do this with the class() function, as the code on the right shows.

2. Vector

Create a vector

In R, you create a vector with the combine function c(). You place the vector elements separated by a comma between the parentheses. For example:

- numeric_vector <- c(1, 2, 3)
- character_vector <- c("a", "b", "c")

Once you have created these vectors in R, you can use them to do calculations.

Naming a vector

You can give a name to the elements of a vector with the names() function. Have a look at this example:

- some_vector <- c("John Doe", "poker player")
- names(some_vector) <- c("Name", "Profession")

This code first creates a vector some_vector and then gives the two elements a name. The first element is assigned the name Name, while the second element is labeled Profession. Printing the contents to the console yields following output:

```
      Name     Profession
"John Doe" "poker player"
```

Calculating total winnings

It is important to know that if you sum two vectors in R, it takes the element-wise sum. For example, the following three statements are completely equivalent:

- c(1, 2, 3) + c(4, 5, 6)
- c(1 + 4, 2 + 5, 3 + 6)
- c(5, 7, 9)

You can also do the calculations with variables that represent vectors:

- a <- c(1, 2, 3)
- b <- c(4, 5, 6)
- c <- a + b

Total winnings?

A function that helps you to answer this question is sum(). It calculates the sum of all elements of a vector. For example, to calculate the total amount of money you have lost/won with poker you do:

- total_poker <- sum(poker_vector)

Comparing total winnings

you can use total_poker > total_roulette

Vector selection

To select elements of a vector (and later matrices, data frames, ...), you can use square brackets. Between the square brackets, you indicate what elements to select. For example, to select the first element of the vector, you type `poker_vector[1]`. To select the second element of the vector, you

type `poker_vector[2]`, etc. Notice that the first element in a vector has index 1, not 0 as in many other programming languages.

To select multiple elements from a vector, you can add square brackets at the end of it. You can indicate between the brackets what elements should be selected. For example: suppose you want to select the first and the fifth day of the week: use the vector `c(1, 5)` between the square brackets. For example, the code below selects the first and fifth element of `poker_vector`:

```
poker_vector[c(1, 5)]
```

Calculate the average of the values in `poker_start` with the **`mean()`** function. Simply print out the result so you can inspect it.

Selection by comparison - Step 1

By making use of comparison operators, we can approach the previous question in a more proactive way.

The (logical) comparison operators known to R are:

- `<` for less than
- > for greater than
- `<=` for less than or equal to
- `>=` for greater than or equal to
- `==` for equal to each other
- `!=` not equal to each other

As seen in the previous chapter, stating `6 > 5` returns `TRUE`. The nice thing about R is that you can use these comparison operators also on vectors. For example:

```
> c(4, 5, 6) > 5
```

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

This command tests for every element of the vector if the condition stated by the comparison operator is `TRUE` or `FALSE`.

In the previous exercises you used `selection_vector <- poker_vector > 0` to find the days on which you had a positive poker return. Now, you would like to know not only the days on which you won, but also how much you won on those days.

You can select the desired elements, by putting `selection_vector` between the square brackets that follow `poker_vector`:

```
poker_vector[selection_vector]
```

R knows what to do when you pass a logical vector in square brackets: it will only select the elements that correspond to `TRUE` in `selection_vector`.

3. Matrices

What's a matrix?

In R, a matrix is a collection of elements of the same data type (numeric, character, or logical) arranged into a fixed number of rows and columns. Since you are only working with rows and columns, a matrix is called two-dimensional.

You can construct a matrix in R with the `matrix()` function. Consider the following example:

```
matrix(1:9, byrow = TRUE, nrow = 3)
```

In the `matrix()` function:

The first argument is the collection of elements that R will arrange into the rows and columns of the matrix. Here, we use `1:9` which is a shortcut for `c(1, 2, 3, 4, 5, 6, 7, 8, 9)`.

The argument `byrow` indicates that the matrix is filled by the rows. If we want the matrix to be filled by the columns, we just place `byrow = FALSE`.

The third argument `nrow` indicates that the matrix should have three rows.

Naming a matrix

To help you remember what is stored in `star_wars_matrix`, you would like to add the names of the movies for the rows. Not only does this help you to read the data, but it is also useful to select certain elements from the matrix.

Similar to vectors, you can add names for the rows and the columns of a matrix

```
rownames(my_matrix) <- row_names_vector
```

```
colnames(my_matrix) <- col_names_vector
```

Calculating the worldwide box office

In R, the function `rowSums()` conveniently calculates the totals for each row of a matrix. This function creates a new vector:

```
rowSums(my_matrix)
```

Adding a column

In the previous exercise you calculated the vector that contained the worldwide box office receipt for each of the three Star Wars movies. However, this vector is not yet part of `star_wars_matrix`.

You can add a column or multiple columns to a matrix with the `cbind()` function, which merges matrices and/or vectors together by column. For example:

```
big_matrix <- cbind(matrix1, matrix2, vector1 ...)
```

Adding a row

Just like every action has a reaction, every `cbind()` has an `rbind()`. (We admit, we are pretty bad with metaphors.)

Your R workspace, where all variables you defined 'live' (check out what a workspace is), has already been initialized and contains two matrices:

`star_wars_matrix` that we have used all along, with data on the original trilogy,

`star_wars_matrix2`, with similar data for the prequels trilogy.

Type the name of these matrices in the console and hit Enter if you want to have a closer look. If you want to check out the contents of the workspace, you can type `ls()` in the console.

The total box office revenue for the entire saga

Just like **`cbind()`** has **`rbind()`**, **`colSums()`** has **`rowSums()`**. Your R workspace already contains the `all_wars_matrix` that you constructed in the previous exercise; type `all_wars_matrix` to have another look. Let's now calculate the total box office revenue for the entire saga.

Selection of matrix elements

Similar to vectors, you can use the square brackets `[ ]` to select one or multiple elements from a matrix. Whereas vectors have one dimension, matrices have two dimensions. You should therefore use a comma to separate the rows you want to select from the columns. For example:

`my_matrix[1,2]` selects the element at the first row and second column.

`my_matrix[1:3,2:4]` results in a matrix with the data on the rows 1, 2, 3 and columns 2, 3, 4.

If you want to select all elements of a row or a column, no number is needed before or after the comma, respectively:

`my_matrix[,1]` selects all elements of the first column.

`my_matrix[1,]` selects all elements of the first row.

A little arithmetic with matrices

Just like `2 * my_matrix` multiplied every element of `my_matrix` by two, `my_matrix1 * my_matrix2` creates a matrix where each element is the product of the corresponding elements in `my_matrix1` and `my_matrix2`.

After looking at the result of the previous exercise, big boss Lucas points out that the ticket prices went up over time. He asks to redo the analysis based on the prices you can find in `ticket_prices_matrix` (source: imagination).

Those who are familiar with matrices should note that this is not the standard matrix multiplication for which you should use `%*%` in R.

4. Factors

What's a factor and why would you use it?

In this chapter you dive into the wonderful world of **factors**.

The term factor refers to a statistical data type used to store categorical variables. The difference between a categorical variable and a continuous variable is that a categorical variable can belong to a **limited number of categories**. A continuous variable, on the other hand, can correspond to an infinite number of values.

It is important that R knows whether it is dealing with a continuous or a categorical variable, as the statistical models you will develop in the future treat both types differently. (You will see later why this is the case.)

A good example of a categorical variable is sex. In many circumstances you can limit the sex categories to "Male" or "Female". (Sometimes you may need different categories. For example, you may need to consider chromosomal variation, hermaphroditic animals, or different cultural norms, but you will always have a finite number of categories.)

To create factors in R, you make use of the function `factor()`. First thing that you have to do is create a vector that contains all the observations that belong to a limited number of categories. For example, `sex_vector` contains the sex of 5 different individuals:

```
sex_vector <- c("Male","Female","Female","Male","Male")
```

It is clear that there are two categories, or in R-terms **'factor levels'**, at work here: "Male" and "Female".

The function `factor()` will encode the vector as a factor:

```
factor_sex_vector <- factor(sex_vector)
```

There are two types of categorical variables: a **nominal categorical variable** and an **ordinal categorical variable**.

A nominal variable is a categorical variable without an implied order. This means that it is impossible to say that 'one is worth more than the other'. For example, think of the categorical variable `animals_vector` with the categories `"Elephant"`, `"Giraffe"`, `"Donkey"` and `"Horse"`. Here, it is impossible to say that one stands above or below the other. (Note that some of you might disagree).

In contrast, ordinal variables do have a natural ordering. Consider for example the categorical variable `temperature_vector` with the categories: `"Low"`, `"Medium"` and `"High"`. Here it is obvious that `"Medium"` stands above `"Low"`, and `"High"` stands above `"Medium"`.

Animals

- animals_vector <- c("Elephant", "Giraffe", "Donkey", "Horse")
- factor_animals_vector <- factor(animals_vector)
- factor_animals_vector

Temperature

- temperature_vector <- c("High", "Low", "High","Low", "Medium")
- factor_temperature_vector <- factor(temperature_vector, order = TRUE, levels = c("Low", "Medium", "High"))
- factor_temperature_vector

Summarizing a factor

After finishing this course, one of your favorite functions in R will be `summary()`. This will give you a quick overview of the contents of a variable:

```
summary(my_var)
```

Going back to our survey, you would like to know how many `"Male"` responses you have in your study, and how many `"Female"` responses. The `summary()` function gives you the answer to this question.

`speed_vector` should be converted to an ordinal factor since its categories have a natural ordering. By default, the function `factor()` transforms `speed_vector` into an unordered factor. To create an ordered factor, you have to add two additional arguments: `ordered` and `levels`.

```
factor(some_vector,
       ordered = TRUE,
       levels = c("lev1", "lev2" ...))
```

By setting the argument `ordered` to `TRUE` in the function `factor()`, you indicate that the factor is ordered. With the argument `levels` you give the values of the factor in the correct order.

Comparing ordered factors

The fact that `factor_speed_vector` is now ordered enables us to compare different elements (the data analysts in this case). You can simply do this by using the well-known operators.

- Use `[2]` to select from `factor_speed_vector` the factor value for the second data analyst. Store it as `da2`.
- Use `[5]` to select the `factor_speed_vector` factor value for the fifth data analyst. Store it as `da5`.
- Check if `da2` is greater than `da5`; simply print out the result. Remember that you can use the `>` operator to check whether one element is larger than the other.

5. Data frames

What's a data frame?

You may remember from the chapter about matrices that all the elements that you put in a matrix should be of the same type. Back then, your data set on Star Wars only contained numeric elements.

When doing a market research survey, however, you often have questions such as:

'Are you married?' or 'yes/no' questions (`logical`)

'How old are you?' (`numeric`)

'What is your opinion on this product?' or other 'open-ended' questions (`character`)

...

The output, namely the respondents' answers to the questions formulated above, is a data set of different data types. You will often find yourself working with data sets that contain different data types instead of only one.

A data frame has the variables of a data set as columns and the observations as rows. This will be a familiar concept for those coming from different statistical software packages such as SAS or SPSS.

Quick, have a look at your data set

Working with large data sets is not uncommon in data analysis. When you work with (extremely) large data sets and data frames, your first task as a data analyst is to develop a clear understanding of its structure and main elements. Therefore, it is often useful to show only a small part of the entire data set.

So how to do this in R? Well, the function `head()` enables you to show the first observations of a data frame. Similarly, the function `tail()` prints out the last observations in your data set.

Both `head()` and `tail()` print a top line called the 'header', which contains the names of the different variables in your data set.

Have a look at the structure

Another method that is often used to get a rapid overview of your data is the function `str()`. The function `str()` shows you the structure of your data set. For a data frame it tells you:

- The total number of observations (e.g. 32 car types)
- The total number of variables (e.g. 11 car features)
- A full list of the variables names (e.g. `mpg`, `cyl` ...)
- The data type of each variable (e.g. `num`)
- The first observations

Applying the `str()` function will often be the first thing that you do when receiving a new data set or data frame. It is a great way to get more insight in your data set before diving into the real analysis.

Creating a data frame

Since using built-in data sets is not even half the fun of creating your own data sets, the rest of this chapter is based on your personally developed data set. Put your jet pack on because it is time for some space exploration!

As a first goal, you want to construct a data frame that describes the main characteristics of eight planets in our solar system. According to your good friend Buzz, the main features of a planet are:

- The type of planet (Terrestrial or Gas Giant).
- The planet's diameter relative to the diameter of the Earth.
- The planet's rotation across the sun relative to that of the Earth.
- If the planet has rings or not (TRUE or FALSE).

After doing some high-quality research on Wikipedia, you feel confident enough to create the necessary vectors: `name`, `type`, `diameter`, `rotation` and `rings`; these vectors have already been coded up on the right. The first element in each of these vectors correspond to the first observation.

You construct a data frame with the **`data.frame()`** function. As arguments, you pass the vectors from before: they will become the different columns of your data frame. Because every column has the same length, the vectors you pass should also have the same length. But don't forget that it is possible (and likely) that they contain different types of data.

Use the function **data.frame()** to construct a data frame. Pass the vectors name, type, diameter, rotation and rings as arguments to data.frame(), in this order. Call the resulting data frame planets_df.

Selection of data frame elements

Similar to vectors and matrices, you select elements from a data frame with the help of square brackets `[ ]`. By using a comma, you can indicate what to select from the rows and the columns respectively. For example:

`my_df[1,2]` selects the value at the first row and second column in `my_df`.

`my_df[1:3,2:4]` selects rows 1, 2, 3 and columns 2, 3, 4 in `my_df`.

Sometimes you want to select all elements of a row or column. For example, `my_df[1, ]` selects all elements of the first row. Let us now apply this technique on `planets_df`!

Instead of using numerics to select elements of a data frame, you can also use the variable names to select columns of a data frame.

Suppose you want to select the first three elements of the `type` column. One way to do this is

```
planets_df[1:3,2]
```

A possible disadvantage of this approach is that you have to know (or look up) the column number of `type`, which gets hard if you have a lot of variables. It is often easier to just make use of the variable name:

```
planets_df[1:3,"type"]
```

Only planets with rings

You will often want to select an entire column, namely one specific variable from a data frame. If you want to select all elements of the variable `diameter`, for example, both of these will do the trick:

```
planets_df[,3]
```

```
planets_df[,"diameter"]
```

However, there is a short-cut. If your columns have names, you can use the `$` sign:
`planets_df$diameter`

You probably remember from high school that some planets in our solar system have rings and others do not. Unfortunately you can not recall their names. Could R help you out?

If you type `rings_vector` in the console, you get:

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

This means that the first four observations (or planets) do not have a ring (`FALSE`), but the other four do (`TRUE`). However, you do not get a nice overview of the names of these planets, their diameter, etc. Let's try to use `rings_vector` to select the data for the four planets with rings.

Only planets with rings but shorter

So what exactly did you learn in the previous exercises? You selected a subset from a data frame (`planets_df`) based on whether or not a certain condition was true (rings or no rings), and you managed to pull out all relevant data. Pretty awesome! By now, NASA is probably already flirting with your CV ;-).

Now, let us move up one level and use the function **`subset()`**. You should see the **`subset()`** function as a short-cut to do exactly the same as what you did in the previous exercises.

```
subset(my_df, subset = some_condition)
```

The first argument of **`subset()`** specifies the data set for which you want a subset. By adding the second argument, you give R the necessary information and conditions to select the correct subset.

The code below will give the exact same result as you got in the previous exercise, but this time, you didn't need the `rings_vector`!

```
subset(planets_df, subset = rings)
```

Sorting

Making and creating rankings is one of mankind's favorite affairs. These rankings can be useful (best universities in the world), entertaining (most influential movie stars) or pointless (best 007 look-a-like).

In data analysis you can sort your data according to a certain variable in the data set. In R, this is done with the help of the function **`order()`**.

`order()` is a function that gives you the ranked position of each element when it is applied on a variable, such as a vector for example:

```
> a <- c(100, 10, 1000)
> order(a)
[1] 2 1 3
```

10, which is the second element in `a`, is the smallest element, so 2 comes first in the output of `order(a)`. 100, which is the first element in `a` is the second smallest element, so 1 comes second in the output of `order(a)`.

This means we can use the output of `order(a)` to reshuffle `a`:

```
> a[order(a)]
[1]   10  100 1000
```

Sorting your data frame

Alright, now that you understand the `order()` function, let us do something useful with it. You would like to rearrange your data frame such that it starts with the smallest planet and ends with the largest one. A sort on the `diameter` column.

6. Lists

Lists, why would you need them?

Congratulations! At this point in the course you are already familiar with:

Vectors (one dimensional array): can hold numeric, character or logical values. The elements in a vector all have the same data type.

Matrices (two dimensional array): can hold numeric, character or logical values. The elements in a matrix all have the same data type.

Data frames (two-dimensional objects): can hold numeric, character or logical values. Within a column all elements have the same data type, but different columns can be of different data type.

Pretty sweet for an R newbie, right? ;-)

A **list** in R is similar to your to-do list at work or school: the different items on that list most likely differ in length, characteristic, and type of activity that has to be done.

A list in R allows you to gather a variety of objects under one name (that is, the name of the list) in an ordered way. These objects can be matrices, vectors, data frames, even other lists, etc. It is not even required that these objects are related to each other in any way.

You could say that a list is some kind super data type: you can store practically any piece of information in it!

Creating a list

Let us create our first list! To construct a list you use the function `list()`:

```
my_list <- list(comp1, comp2 ...)
```

The arguments to the list function are the list components. Remember, these components can be matrices, vectors, other lists, ...

Creating a named list

Well done, you're on a roll!

Just like on your to-do list, you want to avoid not knowing or remembering what the components of your list stand for. That is why you should give names to them:

```
my_list <- list(name1 = your_comp1,
                name2 = your_comp2)
```

This creates a list with components that are named `name1`, `name2`, and so on. If you want to name your lists after you've created them, you can use the `names()` function as you did with vectors. The following commands are fully equivalent to the assignment above:

```
my_list <- list(your_comp1, your_comp2)
names(my_list) <- c("name1", "name2")
```

Being a huge movie fan (remember your job at LucasFilms), you decide to start storing information on good movies with the help of lists.

Start by creating a list for the movie "The Shining". We have already created the variables `mov`, `act` and `rev` in your R workspace. Feel free to check them out in the console.

Selecting elements from a list

Your list will often be built out of numerous elements and components. Therefore, getting a single element, multiple elements, or a component out of it is not always straightforward.

One way to select a component is using the numbered position of that component. For example, to "grab" the first component of `shining_list` you type

```
shining_list[[1]]
```

A quick way to check this out is typing it in the console. Important to remember: to select elements from vectors, you use single square brackets: `[ ]`. Don't mix them up!

You can also refer to the names of the components, with `[[ ]]` or with the `$` sign. Both will select the data frame representing the reviews:

```
shining_list[["reviews"]]
```

```
shining_list$reviews
```

Besides selecting components, you often need to select specific elements out of these components. For example, with `shining_list[[2]][1]` you select from the second component, `actors` (`shining_list[[2]]`), the first element (`[1]`). When you type this in the console, you will see the answer is Jack Nicholson.

Adding more movie information to the list

Being proud of your first list, you shared it with the members of your movie hobby club. However, one of the senior members, a guy named M. McDowell, noted that you forgot to add the release year. Given your ambitions to become next year's president of the club, you decide to add this information to the list.

To conveniently add elements to lists you can use the `c()` function, that you also used to build vectors:

```
ext_list <- c(my_list , my_val)
```

This will simply extend the original list, `my_list`, with the component `my_val`. This component gets appended to the end of the list. If you want to give the new list item a name, you just add the name as you did before:

```
ext_list <- c(my_list, my_name = my_val)
```