# Data manipulation and visualization with R

## 2.2 Selecting data

Hannes Seller, Jonathan Young (2020)

# 2.2 Selecting data

We learned about the different data types in chapter 1.2. While R implements consistent rules for the selection of data from data containers, each container type has its specific particularities. In this lesson we will look at data selection methods for vectors, matrices, dataframes and lists.

This chapter is rather short in comparison and could have been integrated into the basics. However, this is a topic a beginner might want to read again when the occasion for a certain method of data selection might arise. We therefore kept it short and as an individual chapter.

# Data selection

## Vectors (using indices)

This chapter deals with the selection of individual data from any data container. A selection is usually indicated by square brackets which specify the index or indices or data to be selected.

Please note that R does not use zero-based indexing as many other common languages like Java, JavaScript, C or Python. In R the first index is 1.

Vectors, matrices, DFs and lists have slightly different data selection rules depending on their dimensions and specific data structures.

Vectors use a simple square bracket to select and subset data. Positive indices select data, negative indices exclude data.

```
vector[index]
```

```
# creating a vector of characters
vec <- c( "a", "b", "c", "d", "e", "f")

vec[1] #output: "a"
vec[6] #output: "f"
vec[0] #output: character(0) … an empty vector as there is no index 0

# using a vector of indices to select from a vector
vec[c(1,3,5)] # output: "a" "c" "e" (index 1, 3 and 5)

# using a series to select from a vector
vec[1:3] # output: "a" "b" "c" (index 1 to 3)

# using a vector of different series to select from a vector
vec[c(1:2, 4:5)] # output: "a" "b" "d" "e"  (index 1 to 2 and 4 to 5)

# using an exclusion:
vec[-3] # output: "a" "b" "d" "e" "f"  (all but index 3)
```

2.2 Selecting data | Hannes Seller, Jonathan Young (2020)

# Data selection

## Vectors (using logicals)

Besides indices, we can also use logicals to select data from a vector.

A logical vector (e.g. c(TRUE, FALSE, FALSE)) must have the same length as the vector it is applied to. If the element with index n of the logical vector is TRUE, the element with index n will be selected from the data vector. If FALSE, the data is omitted.

A conditional like "vec > 15" returns TRUE for all elements strictly bigger than 15 or FALSE for all elements smaller than or equal to 15. Applying the resulting logical vector to the data containing vector, all elements will be selected that are strictly bigger than 15.

```
vector[vector condition]
```

```
# creating a vector of integers
vec <- c(4, 8, 15, 16, 23, 42)


# which element is bigger than 15?
vec > 15
# output: TRUE TRUE TRUE TRUE TRUE TRUE


# which element fulfills this condition?
vec[vec > 15]
# output: 16, 23, 42


# which element is at least 15 and smaller than 42?
vec[vec >= 15 & vec < 42]
# output: 15, 16, 23
```

# Data selection

## Vectors (using %in%)

The operator %in% checks whether an element of vector 1 is also an element of vector 2.

The operator also returns a logical vector of TRUE and FALSE indicating for each index whether the element exists in both vectors.

In the example, we have the numbers 1 to 10 as “a” and 5 to 15 as “b”. Naturally, the numbers 5, 6, 7, 8, 9 and 10 exist both in “a” and in “b”.

The condition can be inverted by using !(condition), as seen in the last example, showing only elements that exist in a but not in b.

```
Vector1[vector1 %in% vector2]
```

```
# creating two vectors of integers
a <- c(1:10)       # 1 to 10
b <- (5:15)        # 5 to 15


# which element of a is in b?
a %in% b
FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE


# which element fulfills this condition?
a[a %in% b]
# result: 5  6  7  8  9 10


# which element does not fulfill this condition?
a[!(a %in% b)]
# result: 1 2 3 4
```

# Data selection

## Matrices

With matrices the selection slightly differs as they have two dimensions: [X, Y], where X indicates the row and Y the column.

```
matrix[index row, index_column]
```

The indices are shown next to the matrix, for example “a” has the index [1,1] and “f” [3,2]. The matrix below will be used for the following examples on the side.

```
# creating a matrix with two columns
mat <- matrix(c( "a", "b", "c", "d", "e", "f"), ncol=2)
     [,1] [,2]
[1,] "a"  "d"
[2,] "b"  "e"
[3,] "c"  "f"
```

```
# selecting a single data point
mat[1,2] # output: "d" (row 1, column 2)

# selecting an entire row (by not specifying a column) as vector
mat[1,] # output: "a" "d" (row 1)

# selecting an entire column (by not specifying a row) as vector
mat[,2] # output: "d" "e" "f" (column 2)

# selecting with series
Mat[2:3,1:2] # rows 2 to 3 and columns 1 to 2
# output:
     [,1] [,2]
[1,] "b"  "e"
[2,] "c"  "f"

# using exclusion
Mat[-1,2] # output: "e" "f" (all rows but index 1 and column 2)
```

# Data selection

## Dataframes

The data stored within DFs can be accessed in the same manner as matrices. However, DFs provide more options to select data based on values.

As each column of a DF has a specific name, it can be selected by calling its name using this formula:

```
dataframe$column_name
```

An alternative way is shown below. Using indices with double and single square brackets can be useful when the user loops through multiple columns using an automated script. Nonetheless, the way shown above is generally preferable.

```
dataframe[[column_index]][row_index]
```

For the following examples we use this dataframe:

```
# creating four vectors
patient <- c("A", "B", "C", "D")
id <- c(1213,4681,1348,2347)
height <- c(183,163,177,165)
status <- c(TRUE, FALSE, TRUE, TRUE)

# combining four vectors to a dataframe
df <- data.frame(cbind(patient, id, height, status))

# conversion of height from factor to double *
df$height <- as.double(levels(df$height)[df$height])
```

* see chapter 1.2 for details.

# Data selection

## Dataframes

The example on the side shows different ways to select a specific column in a dataframe.

dataframe$column_name is preferable when the user wants to perform a selection based on the column name. Every column name must be unique (and is forced to be unique, if necessary). Therefore, the name always refers to a specific column.

The dataframe can also be treated as a matrix using dataframe[X,Y] to access individual values or whole columns. This is only useful with small datasets where the indices are easy to handle.

Alternatively the user can treat the dataframe like a list object by calling dataframe[[column_index]] for an individual column or dataframe[[column_index]][row_index] for an individual value.

```
  patient    id height status
1       A  1213    183   TRUE
2       B  4681    163  FALSE
3       C  1348    177   TRUE
4       D  2347    165   TRUE

# selecting column “patient” (factor)
df$patient
df[,1]
df[[1]]
# output: [1] A B C D; Levels: A B C D

# selecting column “height” (double)
df$height
df[,3]
Df[[3]]
# output: [1] 183 163 177 165
```

# Data selection

## Lists (of vectors)

Lists can organize multiple data containers (e.g. vectors, dataframes or even other lists) into one object.

The example on the right combines three vectors into a list. When calling the list, its hierarchy is shown using [[index_list_item]] and [index_data_container]. We have three list items (vectors) each containing three integers.

Accessing the list with list_name[index] returns the desired data still wrapped into a list, while list_name[[index]] only returns the stored data container itself. With list_name[[index]][index] we can access the stored data directly. Even index ranges can be applied.

```
# list of three vectors
a <- c(1,2,3)
b <- c(4,5,6)
c <- c(7,8,9)
d <- list(a,b,c)

# that is how our list looks like:
[[1]]
[1] 1 2 3

[[2]]
[1] 4 5 6

[[3]]
[1] 7 8 9
```

```
# first item of list (still a list)
d[1] # selection with [ ]
[[1]]
[1] 1 2 3

# first item of list (a vector)
d[[1]] # selection with [[ ]]
[1] 1 2 3

# target specific value
d[[1]][2] # selection with [[ ]][ ]
[1] 2

# target specific values
> d[[1]][1:2] # s. with [[ ]][x:y]
[1] 1 2
```

# Data selection

## Lists (of data.frames or matrices)

Dataframes can also be stored in lists. This can be handy when several dataframes were created that do not need to clutter the global environment.

In this example two dataframes were stored within the list “d”. Once the dataframe is accessed via list_name[[index]], the selection with df$column_name becomes available. The returned vector can further be accessed with a vector[ ] selector.

As dataframes can also be treated as matrices, the selection with [row, column] is another option to select specific data. They will be returned as a reduced dataframe if entire rows or columns are selected. If only a single value is wanted, the selection returns this value as a length 1 vector.

```
# list of three dataframes
a <- c(1,2,3)
b <- c(4,5,6)
c <- c(7,8,9)
df_ab <- as.data.frame(cbind(a,b))
df_ac <- as.data.frame(cbind(a,c))
d <- list(df_ab, df_ac)

[[1]]
  a b
1 1 4
2 2 5
3 3 6

[[2]]
  a c
1 1 7
2 2 8
3 3 9
```

```
# access with column name
d[[1]]$a # first df, column a
[1] 1 2 3

# column name and index
d[[1]]$a[2]
[1] 2

# first item, first column / row
d[[1]][ ,1]        d[[1]][1, ]
  a                  a b
1 1                1 1 4
2 2
3 3

# first item of list (still a list)
D[[1]][1,2]
[1] 2
```

# 2.2 Selecting data

## What we learned

In this chapter we learned about the particularities of selecting specific pieces of data from data containers.

As a rule of thumb: the [square brackets] always indicate data selection methods. If multiple hierarchies exist in the data container, e.g. in lists, we can use [[double square brackets]] for the highest level and [regular square brackets] for each subordinate level.

In many cases, dataframes are a better container type than matrices because they offer more options to access specific parts of the data within.

Please come back to this chapter when you need to refresh your knowledge about data selection methods. We kept this chapter short and concise for this purpose.

In the next chapter we will look at ways to manipulate data using the “dplyr” package.