R Tutorial at the WZB

4 - R Basics III

Markus Konrad

November 15, 2018

Introductory note

For this session there are no slides, because I'm not at the WZB to hold a presentation. Instead I provide you with a full document for self-study. As always, there are some tasks at the end of the document which you should complete. I recommend that you not only read the document, but also try out some of the code examples and experiment with them.

Solution for tasks #3

The solutions for tasks #3 are now online on https://wzbsocialsciencecenter.github.io/wzb_r_tutorial/ (https://wzbsocialsciencecenter.github.io/wzb_r_tutorial/).

1 Subsetting vectors

Subsetting is a fundamental technique in processing data. With it, you can specify some criteria to form a subset of some input data. You can also call it "filtering", since your criteria work like a filter that only allow certain elements to be pass that filter.

Imagine this input data: `-3, 12, 1, -1, 9`. After subsetting, the output data is `12, 1, 9`. Now guess, what is the criterion used for subsetting?

As you probably guessed, the criterion was that the subset shall only consist of positive numbers. In the subsection "Subsetting with logical expressions", you'll learn how to do that.

There are two main types of subsetting in R: One is to subset via index, i.e. the position of the element in its data structure. An example might be to select the second element of a vector or a sequence of elements from the fourth to the sixth position. This is called *indexing*. The other type is subsetting via logical expressions. With this, you can specify filter criteria based on a comparison or any other logical expression that you've learned in the previous session. One example of this was subsetting for only positive numbers as shown before.

At the beginning, we will learn how to subset vectors. However, you can also subset data frames to filter observations in them. We will have a short look into this but then concentrate on the “modern R” way for subsetting data frames via the package *dplyr* in a later section about the “R tidyverse”.

1.1 Indexing a single element

Let’s assume we have a vector `x` with the following elements:

```
x <- c('a', 'b', 'c', 'd', 'e')
```

In R, each of the elements in the vector can be accessed by its *index* which is a number in the range \([1, N]\) where \(N\) is the length of the vector. For the simple vector above, we could count the elements. However, for larger data you probably don’t want to do that, so remember that there’s a function to find out the length of a vector:

```
length(x)
```

```
## [1] 5
```

So our vector `x` has the indices 1 to 5 with which we can access its elements. The following table illustrates the concept:

index	x
1	a
2	b
3	c
4	d
5	e

You can also see the indices of a vector at the output in the console. You probably already noticed the `[1]` every time you print a vector to console:

```
x
```

```
## [1] "a" "b" "c" "d" "e"
```

The `[i]` denotes the starting index *i* of the vector in that row of the output. If you have a larger output that spans multiple rows, you can see that in front of each row, the starting index is written in the brackets. So here I produce 30 random numbers from a normal distribution and you can see that in the second row the index starts with 6 and goes to 10, in the third row it starts with 11 and goes to 15 and so on:

```
rnorm(30)
```

```
##  [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
##  [6]  1.71506499  0.46091621 -1.26506123 -0.68685285 -0.44566197
## [11]  1.22408180  0.35981383  0.40077145  0.11068272 -0.55584113
## [16]  1.78691314  0.49785048 -1.96661716  0.70135590 -0.47279141
## [21] -1.06782371 -0.21797491 -1.02600445 -0.72889123 -0.62503927
## [26] -1.68669331  0.83778704  0.15337312 -1.13813694  1.25381492
```

The fundamental subsetting operator in R is `[...]` (square brackets). We can use it to get an element from `x` by its index. Let’s say we want to retrieve the second element. We can do so like this:

```
x[2]
```

```
## [1] "b"
```

The fifth (and last) element would be:

```
x[5]
```

```
## [1] "e"
```

Now what happens when we go beyond the bounds of the vector, i.e. above index 5 or below index 1?

Exceeding the upper bound of the vector results in an `NA` value:

```
x[6]
```

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

However, exceeding the lower bounds results in some special behaviour. Indexing vector at 0 always returns an empty vector of the same type:

```
x[0]
```

```
## character(0)
```

And negative indices produce another interesting behaviour that we'll study in the subsection "Indexing with negative integers".

Note: There is also the double square brackets subsetting operator `[[...]]` but we don't need that for now.

1.2 Indexing multiple elements

What if you want to retrieve multiple elements from a vector, not just one? Luckily, you can also pass a *vector of indices* to the subsetting operator, effectively selecting multiple elements at once. So if we want to select the second and fourth element of `x` we can do so by passing the indices `c(2, 4)`:

```
x[c(2, 4)]
```

```
## [1] "b" "d"
```

This way of subsetting is often used together with sequences, which we introduced in the previous session. Remember that you can specify a sequence of integers using the `A:B` syntax:

```
2:5
```

```
## [1] 2 3 4 5
```

Now we can combine this with the subsetting operator to retrieve all elements from the second to the fifth position:

```
x[2:5]
```

```
## [1] "b" "c" "d" "e"
```

Of course you can also use the `seq()` function to specify a more complex sequence of indices.

1.3 Indexing with negative integers

The previous command gave us all but the first element by specifying a range of indices to select. However, if we want to *exclude* certain indices, we can also pass negative integers as indices. So “-1” means “exclude index 1”:

```
x[-1]
```

```
## [1] "b" "c" "d" "e"
```

This selects all but the first element. You can also pass a vector of negative integers to exclude multiple indices:

```
x[c(-1, -5)]  # exclude indices 1 and 5
```

```
## [1] "b" "c" "d"
```

```
x[-c(1, 5)]   # this is the same as above (we moved the "-" sign before the indexing vector)
```

```
## [1] "b" "c" "d"
```

```
x[-(1:3)]     # you can also combine this with sequences but watch out to set the parantheses!
```

```
## [1] "d" "e"
```

1.4 Subsetting with logical expressions

Instead of declaring the positions (indices) of the elements that you want to select, you can also pass a logical vector to the subsetting operator in order to specify which elements you want to include into your result set, and which not. Only those elements will be selected, for which the logical vector has a `TRUE` element at the same position.

Imagine that we have a logical vector `y` of the same length as `x`:

```
y <- c(TRUE, FALSE, FALSE, TRUE, FALSE)
```

Again, we can construct a table with the vector `x` along with our logical vector `y`:

x	y
a	TRUE
b	FALSE
c	FALSE
d	TRUE
e	FALSE

When using the logical vector `y` for subsetting, only those elements in `x` are selected with a respective `TRUE` value on the right side:

```
x[y]
```

```
## [1] "a" "d"
```

A small side note: The function `which()` tells us, where a logical vector contains `TRUE` elements. Here, it identifies element 1 and 4 to be equal to `TRUE`:

```
which(y)
```

```
## [1] 1 4
```

With this, we can turn a logical vector for subsetting into a vector of indices for the same purpose (`x[which(y)]` returns the same result).

The concept of "vector recycling" also applies to subsetting if you pass a vector, that has not the same length as the input vector. So for example, if you pass `c(FALSE, TRUE)`, every second element is selected:

```
x[c(FALSE, TRUE)]
```

```
## [1] "b" "d"
```

This is because the vector `c(FALSE, TRUE)` is "recycled" to `c(FALSE, TRUE, FALSE, TRUE, FALSE)` to match the same length as `x`.

The real benefit for subsetting with logical vectors comes with logical expressions. Remember that every logical expression like `a > 10` will return a logical vector. To experiment with this, let's define again some data from our previous session:

```
age <- c(20, 35, 19, 51, 20)
smoker <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
country <- factor(c('USA', 'GB', 'GB', 'DE', 'USA'))
```

If we wanted to select only a certain country, we could do so with a logical expression and directly store it in a logical vector:

```
(only_gb <- country == 'GB')
```

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

We can use this logical vector for subsetting now:

```
age[only_gb]
```

```
## [1] 35 19
```

```
smoker[only_gb]
```

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

```
country[only_gb]
```

```
## [1] GB GB
## Levels: DE GB USA
```

In the result from the last command, we can see that we actually only selected the `country` observations with the value `GB`, however the factor levels still represent all countries of that variable. Let's try another example:

```
age >= 20 & age <= 35
```

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

```
country[age >= 20 & age <= 35]
```

```
## [1] USA GB  USA
## Levels: DE GB USA
```

Here, we directly used a logical expression within the square brackets. You can also subset the same object that you use for logical subsetting. Here, we select all values from `age` that are greater than or equal 30:

```
age[age >= 30]
```

```
## [1] 35 51
```

Since `smoker` is already logical vector, we can use it directly for subsetting and calculate the mean age for smokers in one go:

```
mean(age[smoker])
```

```
## [1] 35.33333
```

1.5 Subsetting and assignment

Whenever you subset a vector, you can also assign new values to this section by using the assignment operator `<-` . Let's consider vector `x` again:

```
x
```

```
## [1] "a" "b" "c" "d" "e"
```

If we wanted to replace the third element with "X", we can subset and then assign it:

```
x[3] <- 'X'
x
```

```
## [1] "a" "b" "X" "d" "e"
```

Of course, you can also select multiple elements and replace them:

```
x[-(2:4)] <- 'Y'
x
```

```
## [1] "Y" "b" "X" "d" "Y"
```

Let's consider the `age` vector again:

```
age
```

```
## [1] 20 35 19 51 20
```

Imagine that we made a mistake during data collection and all ages above age 30 are off by 1 additional year. So we need to select all observations above 30, then substract 1 year from those and re-assign it to the vector:

```
age[age > 30] <- age[age > 30] - 1
age
```

```
## [1] 20 34 19 50 20
```

Sometimes, missing values need to be replaced. This can also be done in a similar fashion:

```
(age <- c(20, 35, NA, NA, 40))
```

```
## [1] 20 35 NA NA 40
```

Here we decide to replace each NA value with the median of the `age` vector:

```
age[is.na(age)] <- median(age, na.rm = TRUE)
age
```

```
## [1] 20 35 35 35 40
```

Please note that substituting missing values is a very complex topic (it's called *imputation*) and the above is only an example, **not** a rule to apply whenever you see missing values!

1.6 Named vectors

We introduced vectors last week as a way to store **one-dimensional data** of a **single type**.

You can assign a name to each element in a vector:

```
(age <- c(alice = 33, bob = 32, charlie = 48))
```

```
##   alice     bob charlie
##      33      32      48
```

The names are added to the attributes of the object (find it out with the `attributes()` function).

You can also get and set the names via the `names(...)` function:

```
smokers <- c(FALSE, TRUE, FALSE)
names(smokers) <- c("alice", "bob", "charlie")
smokers
```

```
##   alice     bob charlie
##   FALSE    TRUE   FALSE
```

```
names(smokers)
```

```
## [1] "alice"   "bob"     "charlie"
```

You see that logical expressions and subsetting are a powerful and fundamental tool during data preparation and analysis. We used it now for simple vectors, but the same principles also apply for data frames as we'll see in the next session.

2 Subsetting data frames

Please note: In *modern* R, you usually don't subset data frames with the `[]`-subsetting operator anymore. This is because with the packages in the *tidyverse*, data filtering and other transformations can be done in a more intuitive and shorter way. We will see how to do this in the next session. Still, it is important to understand how subsetting data frames in "traditional" R works. Anyway, I'll keep this section very brief.

Another note: Whenever you see the a message like `## [ reached getOption("max.print") -- omitted X rows ]` in the output of the following commands, this means that some of the output was shortened because it was too long to display it in the document. This is because the data set that we'll use contains several hundred observations.

We can now apply subsetting also to tabular data as with data frames in order to filter observations.

Subsetting data frames also works with the `[]`-subsetting operator. Because we have a two-dimensional data structure (with rows in the first, and columns in the second dimension), we can define two indices to select a value. The general pattern for this is:

`X[row, column]`

with `X` being a data frame.

Consider the built-in "airquality" data set:

```
airquality
```

```
##     Ozone Solar.R Wind Temp Month Day
## 1      41     190  7.4   67     5   1
## 2      36     118  8.0   72     5   2
## 3      12     149 12.6   74     5   3
## 4      18     313 11.5   62     5   4
## 5      NA      NA 14.3   56     5   5
##  [ reached getOption("max.print") -- omitted 148 rows ]
```

If we want to get the value of a certain cell, we can do so by indexing it. For example, this will give us the value of the second row and third column ("Wind"):

```
airquality[2, 3]
```

```
## [1] 8
```

Instead of the column number, we can also use its name:

```
airquality[2, "Wind"]
```

```
## [1] 8
```

Most of the time, you will not deal with single data cells, but instead with vectors. You can pass a sequence as row- or column indices, too. This selects the first three values of the "Wind" column:

```
airquality[1:3, "Wind"]
```

```
## [1]  7.4  8.0 12.6
```

And here we select the first rows of the first two columns:

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

```
##   Ozone Solar.R
## 1    41     190
## 2    36     118
## 3    12     149
```

You can also just omit one of the dimensions to select columns or rows. However, you should *not* omit the comma as we can see here:

```
airquality[1:3,]   # select first three rows (notice the ",")
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
```

```
airquality[,1:3]  # select first three columns (notice the ",")
```

```
##     Ozone Solar.R Wind
## 1      41     190  7.4
## 2      36     118  8.0
## 3      12     149 12.6
## 4      18     313 11.5
## 5      NA      NA 14.3
## 6      28      NA 14.9
## 7      23     299  8.6
## 8      19      99 13.8
## 9       8      19 20.1
## 10     NA     194  8.6
##  [ reached getOption("max.print") -- omitted 143 rows ]
```

If you omit the comma, it will be treated like column selection, not row selection (as could be expected):

```
airquality[1:3]   # same as above
```

```
##     Ozone Solar.R Wind
## 1      41     190  7.4
## 2      36     118  8.0
## 3      12     149 12.6
## 4      18     313 11.5
## 5      NA      NA 14.3
## 6      28      NA 14.9
## 7      23     299  8.6
## 8      19      99 13.8
## 9       8      19 20.1
## 10     NA     194  8.6
##  [ reached getOption("max.print") -- omitted 143 rows ]
```

Of course, you can also pass a logical vector for subsetting. This makes most sense for filtering observations. Here, for example, we select only those observations from month 6 (June):

```
airquality[airquality$Month == 6, ]
```

```
##    Ozone Solar.R Wind Temp Month Day
## 32    NA     286  8.6   78     6   1
## 33    NA     287  9.7   74     6   2
## 34    NA     242 16.1   67     6   3
## 35    NA     186  9.2   84     6   4
## 36    NA     220  8.6   85     6   5
##  [ reached getOption("max.print") -- omitted 25 rows ]
```

Again, if you select rows, don't forget the comma:

```
airquality[airquality$Month == 6]
## Error in `[.data.frame`(airquality, airquality$Month == 6) : undefined columns selected
```

A common use case involves the `complete.cases()` function, which returns `FALSE` in each row containing any `NA` value, hence "marking" each row as "incomplete" that contains at least one `NA`. You can use this to select only complete observations in your data set:

```
airquality[complete.cases(airquality),]
```

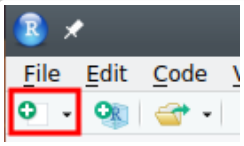
```
##     Ozone Solar.R Wind Temp Month Day
## 1      41     190  7.4   67     5   1
## 2      36     118  8.0   72     5   2
## 3      12     149 12.6   74     5   3
## 4      18     313 11.5   62     5   4
## 7      23     299  8.6   65     5   7
##  [ reached getOption("max.print") -- omitted 106 rows ]
```

3 Working with R scripts

So far, we've only worked with the interactive R console, typing in commands which directly output the results to the console, too. We could also see which commands we've issued by browsing the command history or pressing the `UP` and `DOWN` keys in the console.

This type of working with R is great for learning, for quick investigation or trying things out. It is, however, far from ideal when you're working on a *real* project, which usually involves many commands to be executed in the correct order. In this case, you should record your steps by writing an R **script**.

An R script is nothing but a plain text file with R commands that are executed from top to bottom. It has the file extension `.R`. You can create a new script by clicking on the "plus" icon on the top left in R Studio or selecting "File > New file > R Script" from the main menu.



RStudio: Creating a new file

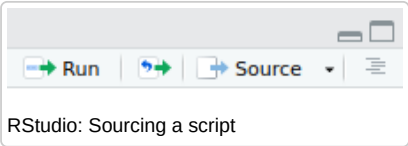
This will open a blank file in what is called the "editor pane". Here you can write your script. If you create or open several files, their names will appear as tabs in the editor pane which allows you to switch between several opened files. An example script would be the following:

```
library(MASS)

data(cats)

female_cats <- cats[cats$Sex == 'F',]
mean_bwt_fem <- mean(female_cats$Bwt)
```

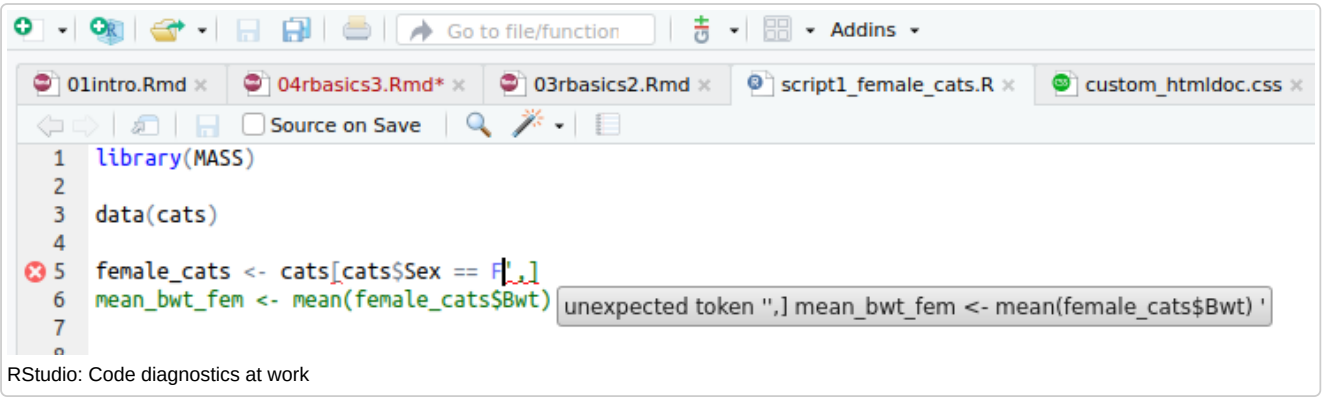
You can then save the script with the keyboard shortcut `CTRL+S`. Once you saved the file, you can execute it as a whole by clicking `Source` (or with `CTRL+SHIFT+S`) on the top right:



RStudio: Sourcing a script

You can also execute individual lines in the console by selecting these lines and clicking "Run" or pressing `CTRL+ENTER`. If no line is selected, the line with the active cursor is executed. Both is very helpful when you are writing scripts. You can write your commands in the script and try them out interactively in the console without copying and pasting.

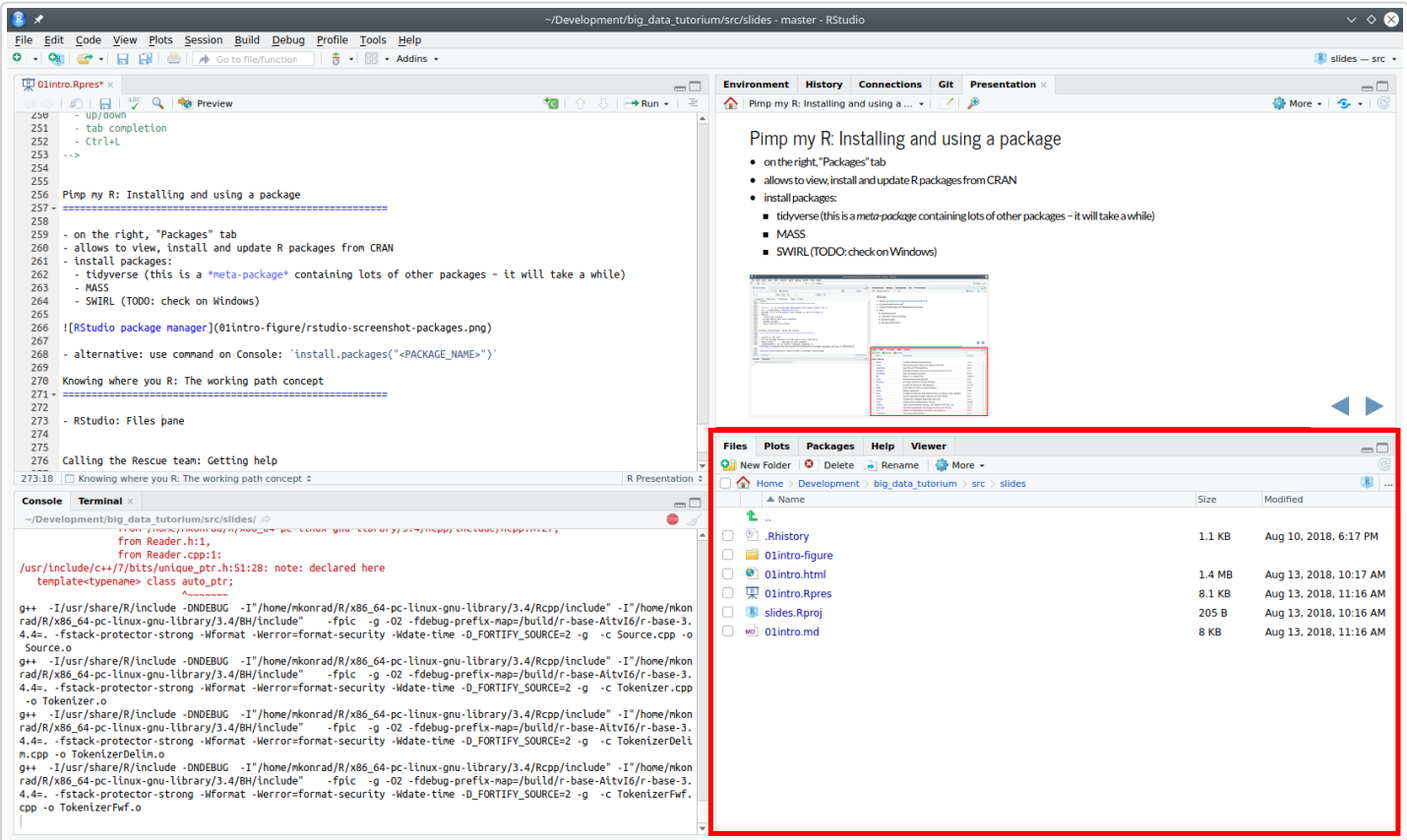
You can use RStudio's auto-completion feature also in your R scripts. That is, you can start to write an object's name (some function, variable, etc.), press `TAB` and the name will automatically be completed (or a list of possible choices will be displayed). This aids typing and helps to avoid mistakes. The "syntax highlighting" feature in the script editor, which highlights keywords, functions and strings in different colors to aids the eye in understanding the code and spotting mistakes. This is also the case for the interactive code diagnostics tool which highlights syntax problems in your code as you type:



RStudio: Code diagnostics at work

As you see, the RStudio script code editor is not just a plain text editor but offers several additional features that help writing code.

Besides creating new script files, you can of course also open existing scripts with "File > Open file…" (or `CTRL+O`) or simply by clicking on a file in the "Files" pane on the lower right in RStudio.



RStudio: Files pane

Scripts are fundamental for **reproducibility in science**. If done correctly, they provide the exact steps from the raw observed data to the research outcome, which is important for other researchers to be able to reproduce your results. But it is also important to the collegues you contribute with and last but not least to yourself, so that you can later recall what you've done to get to your results. But plain code can be hard to understand even when it's written in a clean and well formatted way. To help understanding what's going on in a script, **code comments** are crucial. Remember that everything after a `#` is considered a comment and ignored during code execution. You shouldn't comment every line of code, but you should divide your code into small sections and write a short comment what each section of code is doing.

4 Reading and writing data in different formats

So far, we've only loaded built-in data sets from base R or a package (like the "cats" data set from the package *MASS*), but we never loaded data from a file nor did we ever write results to a file. For a real project however, we need of course both!

Data can be stored to or loaded from files of different formats in R. You may already know that there are sheer endless possibilities when it comes to file formats. There are *plain text formats*, which when opened with a text editor program, are human-readable. Examples of such files are comma-separated values (CSV) files, fixed-width format text files, JavaScript Object Notation (JSON) or XML files. There are also files which cannot be opened with a simple text editor. These are called *binary file formats* and require software like Excel or Stata to open. Luckily, R can handle most of these formats either directly or after loading a package. However, you must first identify the file format so that you know which function from which package you can use to load it into R. If you don't know the file format beforehand, you can identify it most of the time by looking at the file name extension (`.txt`, `.xlsx`, etc.). So it is advisable to turn on file name extension display on your computer (Windows and Mac by default hide the file name extensions). The following table provides an overview about some of the most popular file formats including common file name extensions and additional packages you may need to install and load before being able to work with the respective file format:

Format	Type	File extension(s)	R package required
comma-separated values (CSV)	plain text	.csv	-
fixed-width format	plain text	various (often .txt)	-
JSON	plain text	.json	rjson, jsonlite

Format	Type	File extension(s)	R package required
Excel	binary	.xls, .xlsx	readxl, writexl
Stata	binary	.dta	foreign
SPSS	binary	.sav	foreign

Generally speaking, you'll find packages on CRAN (the package repository for R) for almost every file format. Use R's help system or search online for how to use the package. For two of the most common file formats for data exchange in the Social Sciences, we can now have a deeper look in the next sections.

4.1 Reading and writing CSV files

Please note: The data files for the following sections are contained in the `04rbasics3-resources.zip` file that can be downloaded along with this document from the course website (https://wzbsocialsciencecenter.github.io/wzb_r_tutorial/).

CSV is a plain text file format that stands for comma-separated values, which describes the file format quite good already. Each line can be thought of as table row, and each value on each line is separated by a comma (sometimes instead by a semi-colon). If you opened a CSV file with a text editor, you would see something like this:

```
"city","state","value"
"Bremerhaven","BR",12.1
"Düsseldorf","NRW",25.4
"Bremen","BR",25.7
...
```

Usually, the first line is used to denote the names of the columns of the tabular data. This is called the "header". So in this case, we have three columns: city, state and value. In R, we can use the `read.csv()` function to load such a file. We only need to pass the path to the file that we want to read. But before we can do that, we need to know what our current *working directory* is, and where the file is located in relation to that. I'll copy a slide from the first session here in case you don't remember:

- the *working directory* or *path* is the location on your computer's drive, at which your current R session is working
- reading files, writing files, etc. is **relative to this path**
- finding out the current working path: `getwd()`
- setting the working path: `setwd("<PATH>")`
- **absolute path:** path starts with `/` (MacOS / Unix) or `C:\`
  - depends on your *personal* folder structure
- **relative path:** path starts directly with a file or folder name
  - relative from some other path, e.g. the current working path

So we should at first find out our current working directory with `getwd()`. Now we have two options: 1) We can change our working directory to whereever the data file is located that we want to load. We can do so either with `setwd()` or in RStudio with "Session > Set working directory" from the main menu. 2) We move/copy the data file to our current working directory.

In the following case, I suppose that the file "segindex_sample.csv" exists in a folder "04rbasics3-resources" which is located in our current working directory. We can then load the file with `read.csv()`, assign it to an object and have a look at it:

```
segindex_cities <- read.csv('04rbasics3-resources/segindex_sample.csv')
segindex_cities
```

```
##             city state value
## 1    Bremerhaven    BR  12.1
## 2     Düsseldorf   NRW  25.4
## 3         Bremen    BR  25.7
## 4       Dortmund   NRW  27.4
## 5  Gelsenkirchen   NRW  16.2
## 6      Magdeburg  SANH  22.8
## 7       Freiburg    BW  27.2
## 8        Rostock    MV  39.4
## 9          Mainz    RP  18.0
## 10       Dresden    SN  25.5
```

Using the `str()` function we can see that the data from the CSV file was loaded into a data frame with two factor variables (`city` and `state`) and one numeric variable `value`:

```
str(segindex_cities)
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ city : Factor w/ 10 levels "Bremen","Bremerhaven",..: 2 5 1 3 7 8 6 10 9 4
##  $ state: Factor w/ 7 levels "BR","BW","MV",..: 1 4 1 4 4 6 2 3 5 7
##  $ value: num  12.1 25.4 25.7 27.4 16.2 22.8 27.2 39.4 18 25.5
```

By default, all columns that contain character strings will be converted to factor variables. Most of the time, this is not the desired behavior and you can disable it with setting `stringsAsFactors = FALSE`:

```
segindex_cities <- read.csv('04rbasics3-resources/segindex_sample.csv', stringsAsFactors = FALSE)
str(segindex_cities)   # city and state now are character string variables
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ city : chr  "Bremerhaven" "Düsseldorf" "Bremen" "Dortmund" ...
##  $ state: chr  "BR" "NRW" "BR" "NRW" ...
##  $ value: num  12.1 25.4 25.7 27.4 16.2 22.8 27.2 39.4 18 25.5
```

As already noted, some CSV variables use semi-colons or other characters to seperate the values in each line. You can adjust to this by setting the `sep` parameter to `";"` or any other character. There are numerous other parameters for `read.csv()`, for example a parameter that controls which is considered to be a `NA` value (`na.strings`) or a parameter to control for whether the CSV file has a header (`header` parameter). See the documentation via `help(read.csv)` for more information.

The counterpart of `read.csv()` is called `write.csv()` and it allows you the write a data frame to a CSV file. Let's say we want to filter our data to only include observations from the state "NRW":

```
(segindex_nrw <- segindex_cities[segindex_cities$state == 'NRW',])
```

```
##            city state value
## 2    Düsseldorf   NRW  25.4
## 4      Dortmund   NRW  27.4
## 5 Gelsenkirchen   NRW  16.2
```

We can now export this subset as CSV file. The first parameter for `write.csv` is the data frame that we want to write to file, the second is the path to the CSV file that will be created (or overwritten if it already exists!).

```
write.csv(segindex_nrw, '04rbasics3-resources/segindex_nrw.csv')
```

This will create a new file CSV "segindex_nrw.csv" in the folder "04rbasics3-resources" which must exist in our current working directory. Like with `read.csv()` there are many options for `write.csv()`, for example to adjust the separator character or whether header with the column names should be written to file.

Fixed-width format files are quite similar to CSV files but there the values in a line are not separated by comma or other characters but each column has a certain fixed width. Such files can be read with `read.fwf()` and written with `write.fwf()` respectively.

4.2 Reading and writing Excel files

Files produced with Microsoft Excel or OpenOffice/LibreOffice Calc are used regularly for data exchange in the Social Sciences, although these programs should be used with care (see this Washington Post article for example: An alarming number of scientific papers contain Excel errors (https://www.washingtonpost.com/news/wonk/wp/2016/08/26/an-alarming-number-of-scientific-papers-contain-excel-errors/?noredirect=on&utm_term=.5ef27c2098c2)).

When we want to work with Excel files in R, we can use the *readxl* and *writexl* packages reading and storing data in this file format. If you haven't installed these packages yet, you should so now (either with the RStudio package manager in the lower left pane or with the `install.packages()` function).

The approach is similar to reading and writing CSV files, only the function names differ: You load the file as data frame by specifying a path to `read_xlsx()` and you store a data frame by passing it as first argument to `write_xlsx()`, followed by the path to the file that you want to write the output to. The following lines of code reproduce the same behavior as in the "Reading and writing CSV files" section, this time using Excel files instead:

```
library(readxl)
library(writexl)

segindex_cities <- read_xlsx('04rbasics3-resources/segindex_sample.xlsx')
segindex_nrw <- segindex_cities[segindex_cities$state == 'NRW',]
write_xlsx(segindex_nrw, '04rbasics3-resources/segindex_nrw.xlsx')
```

Note that in contrast to `read.csv()`, `read_xlsx()` does not convert character strings to factors by default.

5 Final notes on R basics

By now, you should have a profound understanding of the R programming language, its basic concepts and data structures. You should know what a vector and a data frame is and how functions can be used to *do* something with them. You should know that there are four different data types for vectors, what kind of data you can store with them and some of the functions and operations that you can apply to them. You should be comfortable in subsetting data with indices and logical expressions. Most important, you should also know how to get help when you got stuck.

So far, the examples were boring I have to admit. We had cats' weight, airquality data and some randomly generated numbers. That's probably far from real Social Science but we will change that gradually in the following sessions. The reason for that is that you need to get things right first with small, clean data, before trying to work with huge, messy real world data sets. But we'll get to it!

6 Tasks

1. Indexing

There are two predefined objects in R that contain all letters from A-Z and a-z, respectively:

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```
letters
```

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

Using numeric indexing (**not** subsetting with logical expressions), try to generate the following output using either `letters` or `LETTERS`:

1. The single letter `"e"`
2. All letters but `"e"`
3. The last five letters in the alphabet (`"v" "w" "x" "y" "z"`)
4. The 23th, 26th and second capital letters (in that order) forming `"W" "Z" "B"`
5. Every second letter starting from 1 (`"a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u" "w" "y"`) Hint: Use the `seq()` function
6. All but the first five letters:
   `"f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"`
7. Create an object `myletters` as a copy of `letters` (`myletters <- letters`). Assign the first five capital letters (from `LETTERS`) to the first five letters of `myletters` so that `myletters` will then contain:
   `"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"`

2. Subsetting with logical expressions

1. Complete **lesson 6 (“Subsetting vectors”)** of SWIRL Course **“R Programming”**. (See the notes in session 2 tasks about installing SWIRL if you have not done that yet.)
2. Consider the following data and subset it according to the mentioned criteria below:

```
retweets <- c(1, 3, 2, 2, 3, 4, 3, 2, 8, 2)
likes <- c(6, 10, 9, 6, 3, 6, 6, 7, 6, 15)
users <- factor(c('WZB_Berlin', 'JWI_Berlin', 'JWI_Berlin', 'gesis_org', 'WZB_Berlin', 'WZB_Berlin',
'WZB_Berlin', 'gesis_org', 'JWI_Berlin', 'WZB_Berlin'))
located_in_berlin <- c(TRUE, TRUE, TRUE, FALSE, TRUE, TRUE, TRUE, FALSE, TRUE, TRUE)
```

Assume that the elements in the vectors are aligned, i.e. the first element in `retweets` corresponds to the first element in `likes` and `users` etc. (as if they were combined in a data frame). Solve all tasks by using logical expressions / logical vectors.

- a. Form subsets of the vectors `retweets` and `likes` to contain only data from the user `WZB_Berlin` .
- b. Form a subset of the vector `users` to contain only elements where `located_in_berlin` is `FALSE` **or** `users` equals `"WZB_Berlin"` (this should return a vector only containing `"gesis_org"` and `"WZB_Berlin"`).
- c. Form subsets of the vectors `retweets` , `likes` and `users` with the criteria to have at least three retweets **and** at least six likes. (Hint: If you want to spare yourself from typing too much, create a logical vector of the criteria at first and re-use it to subset the vectors.)
- d. Calculate the median of `retweets` . Now form a subset of `retweets` , `users` and `located_in_berlin` where retweets are higher than the median.

3. Reading and writing files / subsetting data frames

Create a script file in RStudio that does the following:

1. It loads the CSV file `segindex_sample.csv` (from the accompanying resources file `04rbasics3-resources.zip` available course website (https://wzbsocialsciencecenter.github.io/wzb_r_tutorial/)) into a data frame. Set `read.csv()` to **not** convert strings to factors automatically.
2. It filters this data frame by selecting only observations from the states “NRW”, “RP” and “BW” (Hint: You can use the `%in%` operator for this – it was introduced in the previous session).
3. It saves the filtered data frame to an Excel file `segindex_subset.xlsx` .