

# R Summer Camp 2019

Day 1 - the basic basics

*Geographic Information Centre (GIC), McGill*

*Dr. Tim Elrick, Mon, Aug 26<sup>th</sup>, 2019*

## Contents

<b>Installing the software</b>	<b>2</b>
<b>RStudio</b>	<b>2</b>
<b>Working with RStudio</b>	<b>2</b>
<b>Working with R</b>	<b>3</b>
Arithmetic operators . . . . .	4
Assigning values to objects . . . . .	4
Object naming conventions . . . . .	5
How to get help . . . . .	5
Built-in constants . . . . .	5
Built-in functions . . . . .	6
Logical Operators . . . . .	6
Variable and data types . . . . .	6
<b>Vectors</b>	<b>7</b>
Calculating with vectors . . . . .	8
Indexing vectors . . . . .	8
<b>Building own functions</b>	<b>9</b>
<b>Data structures</b>	<b>10</b>
Matrices . . . . .	10
Creating matrices . . . . .	11
Displaying matrices . . . . .	11
Matrix stats . . . . .	12
Labelling matrices . . . . .	12
Indexing matrices . . . . .	13
Calculating with matrices . . . . .	14
Tibbles (Data frames) . . . . .	14
Using packages . . . . .	15
Creating a tibble . . . . .	15
Indexing tibbles/data frames . . . . .	15
Tibble/Data frame stats . . . . .	16
Working with data frames . . . . .	16
Differences between tibbles and data frames . . . . .	18
<b>Importing &amp; Exporting</b>	<b>18</b>
Excursus: package <code>tidyverse</code> . . . . .	18
Create a table with a spreadsheet . . . . .	19
Read and write <code>.txt</code> -files . . . . .	19
Read and write <code>.csv</code> -files . . . . .	20
Importing and exporting <code>.xls/.xlsx</code> -files . . . . .	20

Importing and exporting <code>.dbf</code> -files . . . . .	21
<b>Saving and loading the workspace</b>	<b>21</b>
<b>Writing nice code</b>	<b>21</b>

---

This handout will guide you through installing R on your machine, understanding how R and RStudio works. You will learn about data and variable types and functions. You will be able to load more functions through packages, import data and retrieve values in a dataset. Of course, you also learn how to save your work.

## Installing the software

As you have seen on the slides, R can do everything. Well, almost everything. It is available free of charge and you can always download the latest version to stay on top of things (as R develops fast).

At [www.r-project.org](http://www.r-project.org) you can download the basic module for the operating system of your choice: Windows, MacOS or Linux derivatives.

To get a bit of the comfort that you are used to when working with other software, we recommend installing an editor. You can use R without one, however, an editor with a graphical user interface (GUI) makes working with R much easier. Here are some popular choices:

- RStudio (<https://www.rstudio.com/>)
- Tinn-R (<https://sourceforge.net/projects/tinn-r/>)
- Notepad++ (<https://notepad-plus-plus.org/>)
- WinEdt with R plug-in ([http://http://www.winedt.com](http://http://www.winedt.com;); <https://rdr.io/cran/RWinEdt/>)

In this course we will use **RStudio**.

## RStudio

In the lower left corner you find the **console**. This is where it's happening, i.e. this is the "R window" to which RStudio sends your code and where it is calculated. You can work in the console only, however, it is more convenient to use the **editor** or *source* (the window in the upper left corner). The upper right corner contains a tab for the **history** of all code executed during a session (in case you want to go back to it) and, most importantly, a tab for the **environment**, where RStudio shows all objects stored in the current session (your *workspace*). In the lower right corner you have multiple tabs to conveniently access the **files** system, all executed **plots** of a session, the R **packages**, the **help** menu and a **viewer**. You can resize all four windows in RStudio. If one is missing just press **Ctrl+Shift+Alt+0**.

## Working with RStudio

Although it is possible to just work in the *console*, it is handier to work in the *editor* for at least three reasons:

1. the editor documents everything you enter, you can check and change the code if necessary;
2. writing your code in the editor helps to detect and correct errors;
3. You can amend your code with comments, which help you keep track of your code when you review it later. You can save the code in the editor as a script. Script files get the file extension `.R`.

If you created objects through your code, they will be stored in the *workspace* and displayed in the *environment* windows in RStudio. These objects are automatically saved in a separate file called `.RData` in the current

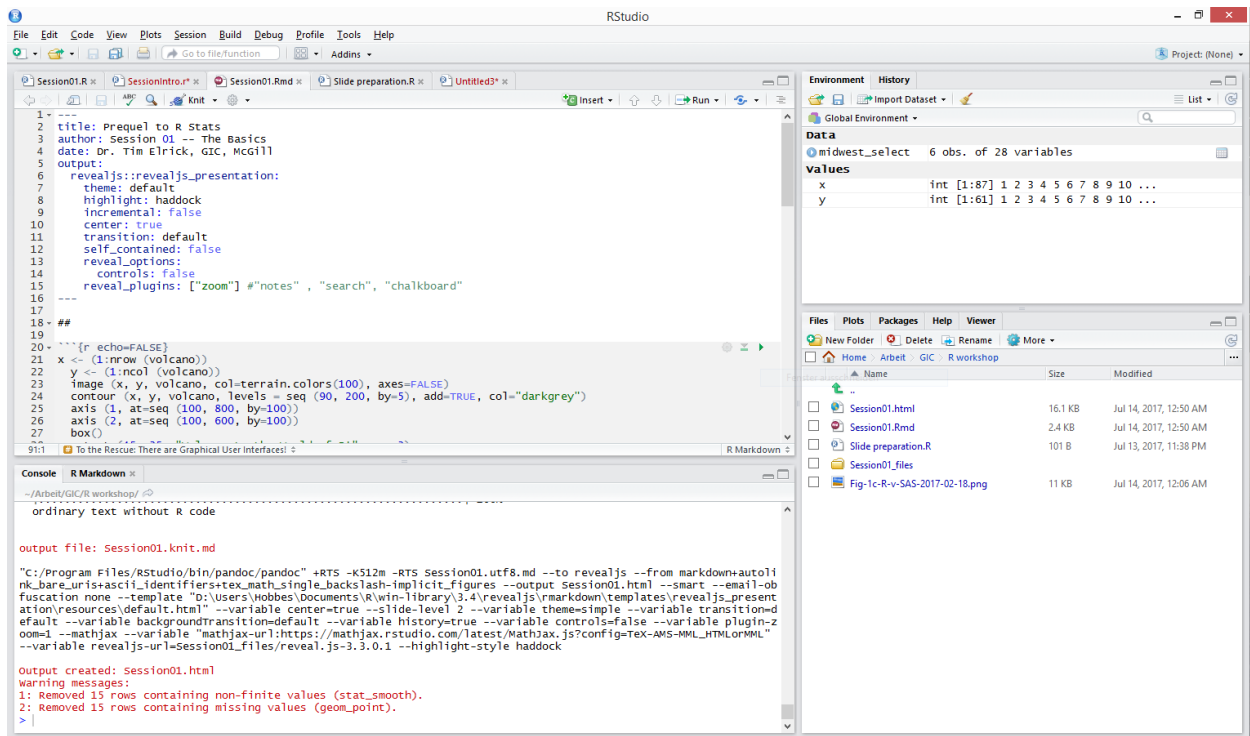



Figure 1: This is the graphical user interface

working directory. You can also save your data separately by clicking the  in the *environment* window. Oh, and how do you find out about your current *working directory*? Go to the *files* tab and choose **More>Go To Working Directory**. When you start working in R, you usually always have to set your working directory first, so that R can find your script and data files.

To make work even more convenient, you can work with *R projects*. R projects save everything including all of your scripts, data and even the working directory in one folder. No need to set a working directory anymore. R projects are saved as `.Rproj`-files in the directory you specified before. For starting a new project go to the



button and select **New project>New directory>New Project**. Then you specify the name of the project and the location where the project should be saved. The tilde (`~`) expresses the current working directory.

## Working with R

So, let's start. We can just use R as a calculator:

```
5 + 3
## [1] 8
```

If you enter this in the editor, nothing happens at first. If you want R to execute your line of code, make sure your cursor is in this line and press **Ctrl+Enter**. Note, that `## [1]` signifies results in this script; in the R console it looks like this: `[1]`.

You can comment your code by using `#` beginning your comment:



```
5 ^ 3 # five to the power of 3
## [1] 125
```

R displays execution results with a [X] before each line. The number in the square brackets represents the index value of each line's first element. We will get back to that later, when we discuss vectors.

When you execute an incomplete line of code, R will indicate this with a + at the beginning of the line; R, then, is waiting for you to complete the code in the *console*:

```
a * (5 + 3
## +
```

Here, of course, you need to add a closing parenthesis ).

When R is still in the process of executing your code it shows a  button in the upper right corner of the *console*; you can interrupt the execution by clicking on the  button or pressing **Esc**.

## Arithmetic operators

R comes with the following basic arithmetic operators:

```
5 + 3 # addition # using the plus sign
## [1] 8
5 - 3 # subtraction # using the minus sign
## [1] 2
5 * 3 # multiplication # using the star sign
## [1] 15
5 / 3 # division # using the slash sign
## [1] 1.666667
5 ^ 3 # exponentiation # using the caret sign
## [1] 125
5 %/% 3 # integer division # using percent slash percent signs
## [1] 1
5 %% 3 # remainder after integer division (modulus) # using percent sign twice
## [1] 2
```

## Assigning values to objects

Now, if we want to use the result in a later analysis we need to store it in an *object*. To do so, we **assign** a number (e.g. 2) or any other expression (e.g. 3+5) to an object; the assignment is done by typing <- or pressing **Alt+-**.

```
a <- 4
result <- 3 + 5
a
## [1] 4
a + result * result
## [1] 68
(a + result) * result
## [1] 96
```

In order to show the value of an object, you must *call* it, i.e. you type the name of the object and execute the code (to execute your code line by line press 'Ctrl+Enter').

Of course, you can use parentheses () as well to override basic mathematical rules.

## Object naming conventions

You can name your objects almost however you like with the following limitations:

- objects have to start with a letter and can be followed by any alpha-numeric character including . (point), - (dash) and \_ (underscore).
- R is case sensitive, i.e. upper and lower case letters are considered to be different (e.g. object `a` is different from object `A`)
- you cannot assign object names that already exist as built-in words, e.g. `for`
- avoid object names that are used by R in existing functions or methods, e.g. `integer`; RStudio indicates already used words by showing them as pop-ups while you write in the editor or console window.
- try to make objects as short as possible (to save you typing), but as long as necessary to understand what values are stored in the object (to help you remember after months coming back to your project)


All-defined objects can be displayed with the following function:

```
ls()
## [1] "a"          "gender"      "hair.color"  "hair.length" "height"
## [6] "result"     "shoe.size"
```

Instead of `ls()` you can also look in RStudio in the *environment* tab, where all assigned object values are displayed.

To clear the *workspace* or *environment* from all self-defined objects you use the bit more complex function:

```
rm(list=ls())
```

... or you just click in the *environment* tab on the  button.

By the way, if R seems to use too much RAM (computational memory of your workstation), you can use `gc()` (for garbage collection) to free up memory.

## How to get help

If you do not know what a function does or just want to find out how it works (incl. parameters), precede the name of the function with a questionmark (`?abs` for lookup the exact function and `??abs` for a vague search) or type it in the help tab in RStudio. Sometimes it is easier to just look for the answer in your favourite search engine in your browser or use the excellent website <https://stackoverflow.com/documentation/r>.

## Built-in constants

Next to the built-in functions R also has a few built-in constants. They are useful e.g. for labelling (see chapter on labelling vectors below):

```
LETTERS # the 26 upper-case letters of the Roman alphabet
## [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 # the 26 lower-case letters of the Roman alphabet
## [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"
month.abb # the three-letter abbreviations for the English month names
## [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
month.name # the English names for the months of the year
## [1] "January" "February" "March" "April" "May"
## [6] "June" "July" "August" "September" "October"
```

```
## [11] "November" "December"  
pi          # the ratio of the circumference of a circle to its diameter  
## [1] 3.141593
```

## Built-in functions

You only get the full functionality of R, when you also use built-in functions, functions of packages and self-built functions (we will get to the latter two later), e.g.

```
sqrt(9)  
## [1] 3  
abs(3 - 5)  
## [1] 2
```

All functions end with parentheses (). In these parentheses you specify the data for the function or any parameters. RStudio handily places your cursor within a set of parentheses, when you press (.

## Logical Operators

R can use not only numerical values (although, it is best at it), but also, for example, logical values, i.e. TRUE and FALSE. You can use these values with the following logical operators:

```
x <- TRUE    # is the same as x <- T  
y <- FALSE   # is the same as y <- F  
x > y        # x is greater than y  
## [1] TRUE  
x >= y       # x is greater than or equal to y  
## [1] TRUE  
x < y        # x is less than y  
## [1] FALSE  
x <= y       # x is less than or equal to y  
## [1] FALSE  
x == y       # x is equal to y  
## [1] FALSE  
x != y       # x is not equal to y  
## [1] TRUE  
!x           # is NOT x  
## [1] FALSE  
x | y        # x OR y  
## [1] TRUE  
x & y        # x AND y  
## [1] FALSE
```

R interprets logical values as numbers when forced to work with logical values in a numerical way: FALSE as 0 (zero) and TRUE as 1. Now you can understand the results (e.g. `x > y`).

## Variable and data types

Usually, the objects store our variables. Therefore, I will use variable instead of object in the remainder. So far, you got to know integer and logical variables; the ones you might use most are **variable types**:

- *character*: a string of signs, letters and numbers
- *integer*: an integer

- *numeric*: any number, including integers and fractions
- *logical*: a boolean value of either TRUE or FALSE

Numbers and logical values can be assigned as is (e.g. `a <- 4` or `x <- FALSE`), any text you have to put in quotation marks (either `a <- "a text"` or `a <- 'another text'`).

For your analysis you usually need a more sophisticated data structure than a simple value-containing variable. R offers the following **data structures**, that we will discuss in detail below:

- *vector*: number of elements of the same variable type
- *matrix*: a table of the same variable type where all columns have the same number of rows (i.e. *length*), i.e. a two-dimensional table
- *array*: a multidimensional table
- *data.frame*: a table that can contain different variable types of the same length
- *tibble*: a more recent form of a data.frame (from the **tibble** package; more on this see session 2)
- *list*: a collection of different variable types of different length
- *factor*: a structure for categorical data of predefined values

## Vectors

The simplest data type is a vector. A vector is a one-dimensional row of elements. The elements can be of any, but the same, variable type. Actually, all variables in R are vectors, even if they only have one element. Vectors of higher dimensions are matrices (2-dimensional vector) or arrays (more than 2 dimensions).

The assignment that we made so far (e.g. `a <- 4`) is a special form of a vector with the length 1, i.e. it contains only one element.

To create a vector with more than one element, we have to concatenate all the elements with the function `c()`, e.g.

```
x <- c(2, 5, 10, 14, 3, 10)
y <- c("a", "bc", NA, "def")
x
## [1]  2  5 10 14  3 10
y
## [1] "a"  "bc" NA  "def"
```

If a value is missing it is signified by the value NA (not available).

To find out about the type of a variable, you can use `class()`, if you want to :

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

Or you can use `str()`, which provides you with the structure of a variable, including its type:

```
str(x)
##  num [1:6] 2 5 10 14 3 10
str(y)
##  chr [1:4] "a" "bc" NA "def"
```

If you want to create an empty vector of a specific type you can do this by using `vector(mode, length)`:

```
num.vec <- vector("numeric", 10) # equivalent to the function numeric(10)
log.vec <- vector("logical", 5)  # equivalent to the function logical(5)
num.vec
```

```
## [1] 0 0 0 0 0 0 0 0 0 0
log.vec
## [1] FALSE FALSE FALSE FALSE FALSE
```

Speaking of length. You can find out about the length of a vector (i.e. the amount of elements stored in it) by using `length()`. Note that a vector can be of length 0 (zero):

```
length(x)
## [1] 6
length(log.vec)
## [1] 5
length(y)
## [1] 4
nchar(y)
## [1] 1 2 NA 3
```

For character vectors `nchar()` gives you the length of all the elements within this vector.

## Calculating with vectors

R is an **object based** programming language. All elements are objects, i.e. vectors as well. This is what makes R so powerful. Let's look, for example, at calculating with vectors.

```
x                # remember we had assigned these numbers to x
## [1] 2 5 10 14 3 10
x * 2            # calculation of a vector with a scalar (i.e. single number)
## [1] 4 10 20 28 6 20
z <- c(1,3,2)    # z is a vector of length 3
x + z            # as x has a length of 6, z is a multiple of x
## [1] 3 8 12 15 6 12
```

The calculation is applied to all elements of the vector (e.g. `x*2`). When you form an operation with two vectors one has to be a multiple of the other; the shorter vector is 'recycled', i.e. applied repeatedly.

Functions also work on vectors, e.g.:

```
sum(x)           # sums all numbers of a numerical vector
## [1] 44
mean(x)          # calculates the arithmetic mean of all numbers of a vector
## [1] 7.333333
min(x)           # looks for the smallest value in a vector
## [1] 2
max(y)           # looks for the largest value in a vector
## [1] NA
quantile(x)      # shows you the quartiles
## 0% 25% 50% 75% 100%
## 2.0 3.5 7.5 10.0 14.0
sort(x)          # sorts the elements of a vector
## [1] 2 3 5 10 10 14
is.na(y)         # checking whether a vector contains missing values
## [1] FALSE FALSE TRUE FALSE
```

## Indexing vectors

To access one or more elements of a vector, you can index the element(s) with square brackets (`[]`):



```
x
## [1] 2 5 10 14 3 10
x[4]                                # calls the 4th element
## [1] 14
x[3:5]                              # calls the 3rd to 5th elements
## [1] 10 14 3
x[c(1,5,2)]                         # calls element 1, 5 and 2
## [1] 2 3 5
x[c(-1,-3)]                         # calls all *but* element 1 and 3
## [1] 5 14 3 10
x[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)] # calls only elements set to TRUE
## [1] 2 10 3
x[7]                                # out of range index
## [1] NA
```

For those of you, who have knowledge of other programming languages, be aware, that R starts indexing with 1 (not 0). Note, that you cannot mix positive and negative indexes (e.g. `c(-1,1)`). An ‘out-of-range’ index will get you the result `NA`, i.e. missing.

Of course, you cannot only retrieve a specific element of a vector, but also assign a value to a specific element:

```
x[3] <- sum(x)
x
## [1] 2 5 44 14 3 10
```

If you want to find the position(s) of certain values, you use `which()`:

```
which(x > 9)
## [1] 3 4 6
which.max(x)      # find the position of the biggest value of a vector
## [1] 3
which.min(x)      # find the position of the smallest value of a vector
## [1] 1
x[which.max(x)]   # get to know the value of the biggest value of a vector
## [1] 44
```

## Building own functions

To build your own function you have to write the actual function in the curly brackets after you assigned a `function(x)`. If you want to use more *external variables* to be entered in the function you can define more *arguments* within the parentheses, e.g. `function(x,y,z)`; this would mean that you have to enter three arguments/values when you call the function. Within the definition of a function you can use as many *internal variables* as you like.

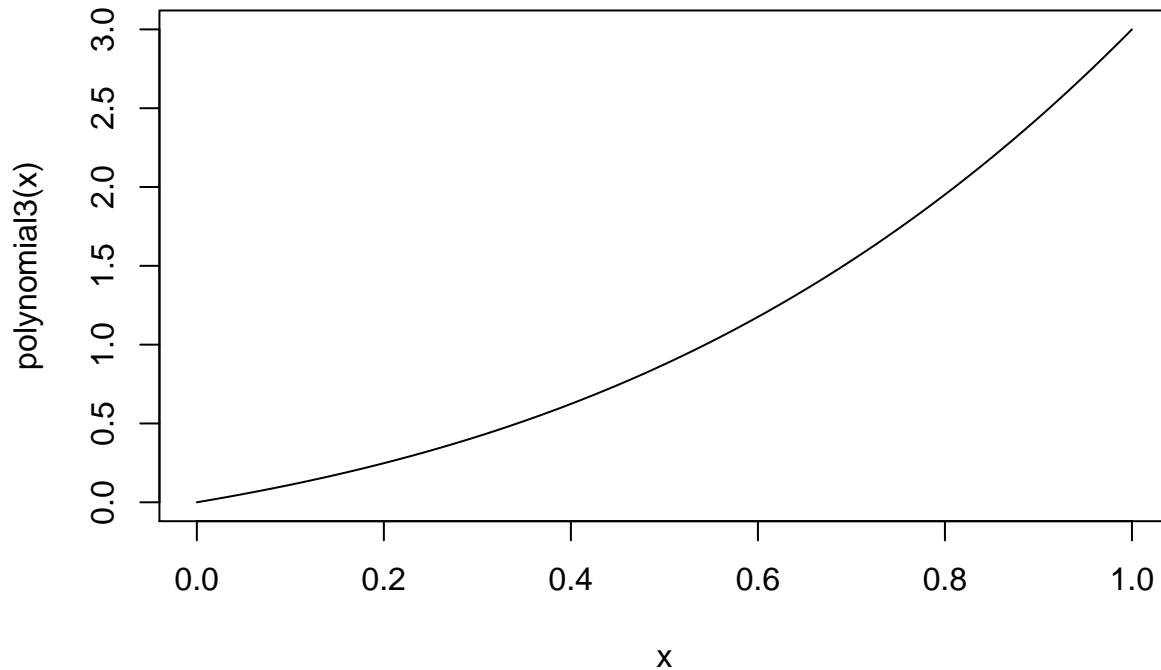
```
polynomial3 <- function(x){
  x + x^2 + x^3
}

# Let's define our own standard deviation function
stand.dev <- function(x, na.remove = TRUE){
  sqrt( sum( (x - mean(x, na.rm = na.remove) )^2 ) / (length(x) - 1) )
}

stand.dev(shoe.size) # now we have to call it with some dataset
```

```
## [1] 2.533114
sd(shoe.size)      # this is the built-in function for comparison
## [1] 2.533114

curve(polynomial3)  # curve(expr) expects a function to be plotted
```



## Data structures

Most dataset are difficult to deal with just as *vectors*. In those case we turn to the other data structure provided in R. We will look at *matrices*, *arrays*, *data frames*, *tibbles*, *lists* and *factors*. Matrices (`matrix()`) can store two dimensions of the same variable type (e.g. `numeric` or `character`), whereas arrays (`array()`) can store as many dimensions as you like (think *3D* or *4D*). Data frames (`data.frame()`) can contain variables of mixed types (this is the usual case for us as scientists and social scientists). Lists (`list()`), however, are the most versatile data structure (but also the most difficult to handle; so, its use is usually avoided). Factors (`factor()`) comprise special data structure for categorial data of predefined values.

### Matrices

A matrix is a two-dimensional data structure that stores data of the same variable type. You can think of a vector as a single row of values or as a single column of values. In this way, you can imagine a matrix as several vectors of the same length combined.

## Creating matrices

You can combine vectors to a matrix by using `rbind()` (each vector represents a new *row* of the matrix) or `cbind()` (each vector represents a new *column* of the matrix). For example, we can combine the following vectors from our class survey:

```
survey.vars.in.rows <- rbind(height, shoe.size, hair.length)
survey.vars.in.cols <- cbind(height, shoe.size, hair.length)
survey.vars.in.rows # assume we have 4 persons participating in the survey
##           [,1] [,2] [,3] [,4]
## height    154.0 182 160.0 178
## shoe.size    3.5   8   8.5   9
## hair.length 50.0   1   1.5   5
survey.vars.in.cols
##           height shoe.size hair.length
## [1,]    154      3.5      50.0
## [2,]    182      8.0       1.0
## [3,]    160      8.5       1.5
## [4,]    178      9.0       5.0
survey <- survey.vars.in.cols # as we will use this more often,
                             # we store in a shorter variable
rm(survey.vars.in.rows, survey.vars.in.cols) # delete variables we do not need anymore
```

If you do not have pre-existing vectors, you can create a new matrix using `matrix(x, nrow=, ncol=, byrow=)`:

```
a <- matrix(1:200, nrow = 20, ncol = 10, byrow = FALSE)
b <- matrix(1:200, nrow = 20, ncol = 10, byrow = TRUE)
```

You do not need to specify both rows and columns (as 200 values e.g. in 20 rows automatically spell out to 10 columns), but it is good practice to do so anyhow. The parameter `byrow=` specifies, as you can see, whether the data is filled into the matrix row by row (`byrow=TRUE`) or column by column (`byrow=FALSE`).

## Displaying matrices

To view the whole matrix use `View()` (*Attention:* capital V). In RStudio this will open a new tab in the editor displaying your data as a spreadsheet (you cannot edit the data in it). As matrices can become pretty big, there are the functions `head()` and `tail()` which only show you the first couple of lines of a matrix in the console:

```
head(a)
##           [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]      1   21   41   61   81  101  121  141  161  181
## [2,]      2   22   42   62   82  102  122  142  162  182
## [3,]      3   23   43   63   83  103  123  143  163  183
## [4,]      4   24   44   64   84  104  124  144  164  184
## [5,]      5   25   45   65   85  105  125  145  165  185
## [6,]      6   26   46   66   86  106  126  146  166  186
head(b)
##           [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]      1     2     3     4     5     6     7     8     9    10
## [2,]     11    12    13    14    15    16    17    18    19    20
## [3,]     21    22    23    24    25    26    27    28    29    30
## [4,]     31    32    33    34    35    36    37    38    39    40
## [5,]     41    42    43    44    45    46    47    48    49    50
```

```
## [6,] 51 52 53 54 55 56 57 58 59 60
tail(a, n = 3) # you can add the parameter n= to specify the number of rows
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [18,] 18 38 58 78 98 118 138 158 178 198
## [19,] 19 39 59 79 99 119 139 159 179 199
## [20,] 20 40 60 80 100 120 140 160 180 200
```

## Matrix stats

To find out about the number of columns and rows of a matrix, you can enquire about its dimensionality by using `dim()`. The first value displays the number of rows, the second one the number of columns; `nrow()` and `ncol()` do this separately:

```
dim(survey)
## [1] 4 3
nrow(survey)
## [1] 4
ncol(survey)
## [1] 3
length(survey) # calculates the number of values in the matrix, i.e. rows times columns
## [1] 12
str(survey) # have also a look at the structure of the matrix
## num [1:4, 1:3] 154 182 160 178 3.5 8 8.5 9 50 1 ...
## - attr(*, "dimnames")=List of 2
## ..$ : NULL
## ..$ : chr [1:3] "height" "shoe.size" "hair.length"
```

## Labelling matrices

To make a matrix more comprehensible, it is possible to label it by using `rownames()` and `colnames()`. Of course, the number of labels should fit the length of columns or rows respectively:

```
rownames(b) <- letters[1:20]
colnames(b) <- LETTERS[1:ncol(b)] # to make your code reusable you can generalize it
head(b, n = 4)
##      A B C D E F G H I J
## a  1 2 3 4 5 6 7 8 9 10
## b 11 12 13 14 15 16 17 18 19 20
## c 21 22 23 24 25 26 27 28 29 30
## d 31 32 33 34 35 36 37 38 39 40
rownames(b) # you can also display the given names
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t"
```

To create labels you sometimes have to concatenate different elements, e.g. 'Pn1' ... 'Pn10' (for participant number 1 to 10). To do so, you use `paste()`:

```
colnames(a) <- paste("Pn", 1:10, sep = "") # 'sep=' defines the separator
# in this case it is set to no space between the elements
head(a, n = 2)
##      Pn1 Pn2 Pn3 Pn4 Pn5 Pn6 Pn7 Pn8 Pn9 Pn10
## [1,] 1 21 41 61 81 101 121 141 161 181
## [2,] 2 22 42 62 82 102 122 142 162 182
```

## Indexing matrices

Same as with vectors, you can index matrices by using square brackets []. Now, you have to insert two values, first the row, then the column index, separated by a comma:

```
b[3, 4] # this calls the element in the 3rd row, 4th column
## [1] 24
```

This corresponds to cell D3 in a common spreadsheet like Microsoft Excel.

If you leave out one of the numbers you get either the full row or full column:

```
b[, 4] # you get all row values of the 4th column
##   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o   p   q   r
##  4  14  24  34  44  54  64  74  84  94 104 114 124 134 144 154 164 174
##   s   t
## 184 194
b[3, ] # you get all column values of the 3rd row
##  A  B  C  D  E  F  G  H  I  J
## 21 22 23 24 25 26 27 28 29 30
```

By indexing a range of numbers, you can build a subset of your matrix:

```
b[3:5, 2:9] # you get only rows 3 to 5 and columns 2 to 9
##      B  C  D  E  F  G  H  I
## c 22 23 24 25 26 27 28 29
## d 32 33 34 35 36 37 38 39
## e 42 43 44 45 46 47 48 49
```

You can use indexing to reorder your matrix:

```
b[, c(10:1)] # by using this index you reverse the order of the columns
##      J  I  H  G  F  E  D  C  B  A
## a  10  9  8  7  6  5  4  3  2  1
## b  20 19 18 17 16 15 14 13 12 11
## c  30 29 28 27 26 25 24 23 22 21
## d  40 39 38 37 36 35 34 33 32 31
## e  50 49 48 47 46 45 44 43 42 41
## f  60 59 58 57 56 55 54 53 52 51
## g  70 69 68 67 66 65 64 63 62 61
## h  80 79 78 77 76 75 74 73 72 71
## i  90 89 88 87 86 85 84 83 82 81
## j 100 99 98 97 96 95 94 93 92 91
## k 110 109 108 107 106 105 104 103 102 101
## l 120 119 118 117 116 115 114 113 112 111
## m 130 129 128 127 126 125 124 123 122 121
## n 140 139 138 137 136 135 134 133 132 131
## o 150 149 148 147 146 145 144 143 142 141
## p 160 159 158 157 156 155 154 153 152 151
## q 170 169 168 167 166 165 164 163 162 161
## r 180 179 178 177 176 175 174 173 172 171
## s 190 189 188 187 186 185 184 183 182 181
## t 200 199 198 197 196 195 194 193 192 191
```

Many times you will generate an index in a variable, and then use this variable to get a subset of your data:

```
g <- which(colnames(b) > "B" & colnames(b) <= "F")
b[, g]
```

```
##      C      D      E      F
## a      3      4      5      6
## b     13     14     15     16
## c     23     24     25     26
## d     33     34     35     36
## e     43     44     45     46
## f     53     54     55     56
## g     63     64     65     66
## h     73     74     75     76
## i     83     84     85     86
## j     93     94     95     96
## k    103    104    105    106
## l    113    114    115    116
## m    123    124    125    126
## n    133    134    135    136
## o    143    144    145    146
## p    153    154    155    156
## q    163    164    165    166
## r    173    174    175    176
## s    183    184    185    186
## t    193    194    195    196
```

## Calculating with matrices

To calculate simple arithmetics with matrices you can use the following functions:

```
rowSums(b) # calculates the sum of the values of each row
##      a      b      c      d      e      f      g      h      i      j      k      l      m      n      o
##     55    155    255    355    455    555    655    755    855    955   1055   1155   1255   1355   1455
##      p      q      r      s      t
##    1555   1655   1755   1855   1955

colSums(b) # calculates the sum of the values of each column
##      A      B      C      D      E      F      G      H      I      J
##    1920   1940   1960   1980   2000   2020   2040   2060   2080   2100

rowMeans(b) # calculates the arithmetic mean of the values of each row
##      a      b      c      d      e      f      g      h      i      j      k      l
##     5.5    15.5    25.5    35.5    45.5    55.5    65.5    75.5    85.5    95.5   105.5   115.5
##      m      n      o      p      q      r      s      t
##    125.5   135.5   145.5   155.5   165.5   175.5   185.5   195.5

colMeans(b) # calculates the arithmetic mean of the values of each column
##      A      B      C      D      E      F      G      H      I      J
##     96     97     98     99    100    101    102    103    104    105
```

## Tibbles (Data frames)

Matrices in R contain *only* numbers or *only* text. The advantage of this accrues in calculation speed: R can process them very fast. As soon as you enter a character value into a numerical matrix all entries will be processed as text. All remaining values cannot be used for calculations any longer:

```
num.mat <- matrix(1:9, ncol = 3)
str(num.mat)
## int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
```

```
num.mat[3, 3] <- "x"
str(num.mat)
## chr [1:3, 1:3] "1" "2" "3" "4" "5" "6" "7" "8" "x"
```

When you want to mix variables of different type, you, therefore, have to use a different data structure, e.g. a `tibble()`. Tibbles are a recent type of data frames. They can contain columns with numerical values and columns with textual values at once. This might be the data structure we most often deal with in our research. To process a data frame, R takes longer (handling only small data structures you will not notice, but the bigger the dataset the more important to think about how you deal with your data).

Before we are able to work with tibbles we have to load a package, as tibbles are not part of the base functionality of R.

### Using packages

To use tibbles you have to use an R package. R packages enhance the capabilities of R quite extensively and according to your immediate needs. You can find all tested R package on the CRAN website: <https://cran.r-project.org/web/packages>. CRAN is the 'Comprehensive R Archive Network', where you find many resources for working with R. To download a package you can either use the function `install.packages()` or you go to the *Packages* tab in the lower right corner of RStudio and click on *Install*. It then appears in the list of installed packages in the *Package* tab. To use a package, it first has to be downloaded into R/RStudio. Then you have to load the package by using `library()`.

```
install.packages("tibble")
library(tibble)
```

As R is evolving quickly, make sure you always update your packages at least when you are updating your R base software using the `update.packages()` function or clicking on 'Update' in the *package tab*.

### Creating a tibble

To build a tibble from different vectors, you just insert them in the `tibble()` function:

```
survey <- tibble(height, shoe.size, hair.color, gender)
head(survey)
## # A tibble: 4 x 4
##   height shoe.size hair.color gender
##   <dbl>    <dbl> <chr>    <chr>
## 1    154      3.5 blonde  female
## 2    182      8   brown   male
## 3    160     8.5 brown   other
## 4    178      9   brown   male
```

### Indexing tibbles/data frames

You can index data frames the same way as matrices and arrays. You can also use column names to address a single variable in the data frame using the `$` (dollar) sign:

```
survey[, 3] # reduces the selection into a vector
## # A tibble: 4 x 1
##   hair.color
##   <chr>
## 1 blonde
```

```
## 2 brown
## 3 brown
## 4 brown
survey$hair.color      # reduces the selection into a vector
## [1] "blonde" "brown" "brown" "brown"
survey[["hair.color"]] # reduces the selection into a vector
## [1] "blonde" "brown" "brown" "brown"
survey["hair.color"]   # keeps the tibble/data frame structure
## # A tibble: 4 x 1
##   hair.color
##   <chr>
## 1 blonde
## 2 brown
## 3 brown
## 4 brown
```

### Tibble/Data frame stats

Data frames are, therefore, two-dimensional data structures with variables as columns and observations or cases as rows.

```
ncol(survey)      # number of variables in data frame
## [1] 4
nrow(survey)      # number of observations in data frame
## [1] 4
length(survey)    # number of variables in data frame (different to matrix!)
## [1] 4
str(survey)       # structure of data frame
## Classes 'tbl_df', 'tbl' and 'data.frame':  4 obs. of  4 variables:
## $ height      : num  154 182 160 178
## $ shoe.size   : num   3.5 8 8.5 9
## $ hair.color  : chr  "blonde" "brown" "brown" "brown"
## $ gender      : chr  "female" "male" "other" "male"
```

### Working with data frames

You index a data frame like a matrix:

```
survey[2:3, ] # display all variables of the observations 2 to 3
## # A tibble: 2 x 4
##   height shoe.size hair.color gender
##   <dbl>   <dbl> <chr>      <chr>
## 1   182     8 brown      male
## 2   160   8.5 brown      other
survey[survey["shoe.size"] >= 8, ] # display all variables of all participants of min. size 8
## # A tibble: 3 x 4
##   height shoe.size hair.color gender
##   <dbl>   <dbl> <chr>      <chr>
## 1   182     8 brown      male
## 2   160   8.5 brown      other
## 3   178     9 brown      male
survey[1:2, "shoe.size"] # display only shoe size of observation 1 and 2
## # A tibble: 2 x 1
```



```
## shoe.size
## <dbl>
## 1 3.5
## 2 8
survey[survey$shoe.size == 10, 3] # displays the hair color of person with shoe size 10
## # A tibble: 0 x 1
## # ... with 1 variable: hair.color <chr>
survey[, -1] # display all variables of observations, but the first variable 'height'
## # A tibble: 4 x 3
## shoe.size hair.color gender
## <dbl> <chr> <chr>
## 1 3.5 blonde female
## 2 8 brown male
## 3 8.5 brown other
## 4 9 brown male
```

This is how you modify a value in a dataframe:

```
#survey[3, 3] <- blonde # Person 3 does have blonde hair
survey[survey$height == 163, "hair.color"] <- "blonde" # is the same as the line above
survey
## # A tibble: 4 x 4
## height shoe.size hair.color gender
## <dbl> <dbl> <chr> <chr>
## 1 154 3.5 blonde female
## 2 182 8 brown male
## 3 160 8.5 brown other
## 4 178 9 brown male
```

The `fix()` function allows to enter or change values in a spreadsheet-like format (this is only advised for smaller data sets), in our case: `fix(df)`.

You can also add a new observation by using `rbind()` or a new variable by using `cbind()`:

```
survey <- rbind(survey, c(146, 5.5, "blonde", "other"))
survey
survey <- cbind(survey,
  "hair.length" = c(10, 2, 2, 5, 70, 3, 15, 10, 10, 1, 120))
survey
```

Deleting variables and observations in a data frame works like this:

```
survey$hair.color <- NA # deletes values of 'hair.color'
head(survey, n = 2)
## # A tibble: 2 x 4
## height shoe.size hair.color gender
## <dbl> <dbl> <lgl> <chr>
## 1 154 3.5 NA female
## 2 182 8 NA male
survey$hair.color <- NULL # deletes the variable 'hair.color'
str(survey)
## Classes 'tbl_df', 'tbl' and 'data.frame': 4 obs. of 3 variables:
## $ height : num 154 182 160 178
## $ shoe.size: num 3.5 8 8.5 9
## $ gender : chr "female" "male" "other" "male"
survey <- survey[, -1] # deletes the first variable ('height')
```

```

survey
## # A tibble: 4 x 2
##   shoe.size gender
##   <dbl> <chr>
## 1     3.5 female
## 2     8   male
## 3     8.5 other
## 4     9   male
survey <- survey[-1, ]      # deletes the first observation
survey
## # A tibble: 3 x 2
##   shoe.size gender
##   <dbl> <chr>
## 1     8   male
## 2     8.5 other
## 3     9   male

```

### Differences between tibbles and data frames

- Displaying: tibble only displays the first 10 rows of the data set and let's you know about the number of missing rows; it also let's you know about all variables, but displays only as many as fit on the screen and tells you about the variable type of the columns.
- StringsAsFactors: a tibble never changes coerces text into factors like a data frame does, if you do not set the parameter.
- Variable names: a tibble never changes variable names; a data frame changes spaces in variable names to dots.
- Column-wise reading: a tibble reads dataset one columns after the other; data frames read the data row-wise.
- Row names: as tibbles consider each row as an observation, it is not possible to assign row names.
- Indexing: a tibble displays a warning when you try to index with a non-existing column name (e.g. `survey.df$gendr`).
- Subsetting: a tibble always returns a tibble when a subset is created (see 'Indexing data frames' above).
- Reading speed: a tibble reads a data set faster than a data frame.

## Importing & Exporting

So far, we created our own data in this workshop. Usually, it is easier to create datasets in other software like Excel or LibreOffice Calc and later import them into R. Further, in your research you most probably want to use pre-existing datasets or created a dataset yourself outside of the R environment that you now want to analyse in R. R provides a wide range of *read* and *write* functions for almost any dataset out there. Many of them are included in the *base* package of R, i.e. you do not have to install any other package. For some of them you have to load a package (e.g. for importing from Excel, Stata, SAS or SPSS).

### Excursus: package tidyverse

Let's briefly talk about a recent development, that started in September 2016: the set of packages assembled under the package name **tidyverse** (<https://www.tidyverse.org/>). The set of packages is designed to help researcher, who use rectangular datasets (i.e. tables as in R data frames and R tibbles), to do a better job and turns out to be very helpful (and, therefore, is used very widely ever since). The **tidyverse** package assembles the following (taken from <https://blog.rstudio.com/2016/09/15/tidyverse-1-0-0/>):

- the core **tidyverse** packages that you are likely to use in almost every analysis:
  - ggplot2, for data visualisation.
  - dplyr, for data manipulation.
  - tidyr, for data tidying.
  - readr, for data import.
  - purrr, for functional programming.
  - tibble, for tibbles, a modern re-imagining of data frames.
- packages for *data manipulation*:
  - hms, for times.
  - stringr, for strings.
  - lubridate, for date/times.
  - forcats, for factors.
- packages for further *data import*:
  - DBI, for databases.
  - haven, for SPSS, SAS and Stata files.
  - httr, for web apis.
  - jsonlite, for JSON.
  - readxl, for .xls and .xlsx files.
  - rvest, for web scraping.
  - xml2, for XML.
- packages for *modelling*:
  - modelr, for simple modelling within a pipeline
  - broom, for turning models into tidy data

To use all of the packages you can download the full set:

```
install.packages("tidyverse")
library(tidyverse)
```

## Create a table with a spreadsheet

If you want to create a new dataset in a spreadsheet like Microsoft Excel, Open/LibreOffice Calc, Apple Numbers or Google Sheets, you should stick to some rules to have less problems importing the data into R:

- Do not format your data (no ‘fix window’, no spanning cells, no named ranges, only one row of variable names, ...).
- Limit data entries to row 2 and higher and column B and up.
- Use column A for row names and row 1 for column/variable names.
- Leave cell A1 empty.
- Replace missing values (i.e. empty cells) by filling them with NA or -9999.
- Do not write comments or text in columns that should contain only numbers.

These rules are valid when you use the tidyverse package to import your data. Stricter rules apply, if you want to use the base R import functions.

## Read and write .txt-files

To import a text file you should first load the **readr** package, and then **read\_table()**. This function reads an ASCII file (i.e. text-based tables) where the variables are separated by spaces. You can and should specify some of the parameters, of which the most used ones are discussed below (for all parameters look up the help **?read\_table**); the parameter values shown are the preset ones and can be changed accordingly.

```
read.table("path/filename.txt",      # insert filename; path if file is not in
          # working directory; note: / is used to
```

```

col_names = TRUE,      # separate folders, not \ as in MS Windows
                        # if the variable names are missing in your
                        # data, you set this to FALSE
col_types = NULL,      # specify variable types, if the automated guess fails
skip = 0,              # skip number of lines before reading data
n_max = Inf,           # set the max number of data rows to be read
na = "NA"              # missing values are set to NA
)

```

The imported data is then stored as a *tibble*.

To export R data to a text file use `write_table(x, "path/filename")` where `x` is the variable you want to export. Note, that you can only export one variable at a time.

## Read and write .csv-files

You can also use the functions from the `readr` package (which is part of the `tidyverse` bundle) to import .csv- and .tsv-files into *tibbles*:

```

library(readr)
read_csv("filename")    # variables separated by ",", decimal separator is "."
read_csv2("filename")   # variables separated by ";", decimal separator is ","
read_tsv("filename")    # variables separated by tabulators
write_csv(x, "filename")

```

Using the `readr` package functions is much faster than the `base` package functions and a progress bar is shown (for big files). There are many other read and write functions in the `readr` package.

## Importing and exporting .xls/.xlsx-files

For *importing* from Excel you can either use the method described for .txt-files or you can use the `readxl` package (part of `tidyverse`) that can directly import Excel files into R without installing other software, as it uses the R built-in C++ library.

If not loaded, you have to run the function first.

Now you can import an .xls/.xlsx-file with `read_excel("path/filename", sheet=, ...)`, where the function auto-detects the format from the extension. If the parameter `sheet=` is not specified, it reads the first sheet of the file. As the `readr` package is part of the `tidyverse` bundle, the imported data structure is a *tibble*.

```

my.data <- read_excel("readxl_example.xlsx")
my.data
my.data <- read_excel("readxl_example.xlsx", sheet = 3, skip = 2) # reads the 3rd sheet
                        # and skips the first 2 rows (that do not contain relevant data)
my.data

```

You can also specify the range of data you want to import by using the parameter `range=` where the range is specified in typical Excel form, e.g. "B4:E34", even including the sheet, e.g. "mtcars!B3:D10". The parameter `col_types=` is the same as `ColClasses` for the `read.table()` function (see above).

Now, the *exporting* is more tricky, as the `readxl` package is only that: it reads Excel files; it does not write them. The easiest way is to save your R data as a .csv- or .txt-file and then import this into Excel.

## Importing and exporting .dbf-files

You might wonder why we list the old database format `dbase` here. However, you might want to use the following functions if you work with ESRI ArcGIS and QGIS shapefiles. One of the multiple files produced by a shapefile is a .dbf-file that stores all the attribute information. For accessing the file you need to load the `foreign` package.

```
library(foreign)
read.dbf("filename", as.is = TRUE) # as.is=TRUE takes care of text being import as 'character'
                                   # and not as 'factor'
write.dbf(x, "filename")           # where x is data frame (!) you want to export
```

This will return a *data frame*. You can convert the data frame into a tibble by using `as.tibble()`.

## Saving and loading the workspace

At the end of a work session, you probably want to save your work. To save all variables of a workspace (i.e. the environment) you use `save.image("path/filename.RData")`. However, if you have set up an R project you can also just close RStudio and the software will save the workspace as well as the scripts (i.e. the code in the editor).

To continue working with a previously saved workspace you use `load("path/filename.RData")`, unless again, you use an R project; then you just open RStudio and continue working.

## Writing nice code

At the end of your first R coding session, I would like to suggest to follow the ‘Google’s R Style Guide’. It is a guide written jointly by the users of the Google R forum of what they agreed on would be ‘good coding practice’. If you follow these guidelines

- you will please your course instructor
- you will be able to interact with other R users more efficiently
- most importantly, you will be able to understand your own code easier when you look at it a week, a semester or years later

The guidelines can be accessed

- as a website: <https://google.github.io/styleguide/Rguide.xml>
- as a presentation: [https://rstudio-pubs-static.s3.amazonaws.com/149679\\_0f07bbb45e5c4f3887d64c0152e0b747.html](https://rstudio-pubs-static.s3.amazonaws.com/149679_0f07bbb45e5c4f3887d64c0152e0b747.html)

If you want to dig deeper into what to do to write ‘good code’, check out the Nicier R Blog (<https://nicercode.github.io/about>) and especially their post on ‘Why I want to write nice R code’: <https://nicercode.github.io/blog/2013-04-05-why-nice-code/>.