# Data Science mit R

Handout

INWT Statistics

Herman Hollerith Zentrum

Böblingen

Juni 2018

INWT Statistics GmbH
Hauptstraße 8
10827 Berlin
+49 30 12082310
info@inwt-statistics.de
www.inwt-statistics.de

INWT Statistics

# Contents

# 1 Introduction to R

## 1.1 What exactly is R?

R is a language and environment for statistical computing and graphics. It is a GNU project which is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. [. . . ]

R provides a wide variety of statistical [. . . ] and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity.

One of R's strengths is the ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed. Great care has been taken over the defaults for the minor design choices in graphics, but the user retains full control.

R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS.

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

The term 'environment' is intended to characterize it as a fully planned and coherent system, rather than an incremental accretion of very specific and inflexible tools, as is frequently the case with other data analysis software.

R, like S, is designed around a true computer language, and it allows users to add additional functionality by defining new functions. [. . . ] Advanced users can write C code to manipulate R objects directly.

Many users think of R as a statistics system. We prefer to think of it of an environment within which statistical techniques are implemented. R can be extended (easily) via packages. There are about eight packages supplied with the R distribution and many more are available through the CRAN family of Internet sites covering a very wide range of modern statistics. [. . . ]

*Source:* www.r-project.org

**Advantages (+) and disadvantages (-) of `R`:**

- [-] interpreter language
- [+] for free and open source
- [+] fast and slim
- [+] always up-to-date
- [+] vector-orientated
- [+] broad range of functions
- [+] adaptability (e.g., for generating graphics)
- [+] interaction with databases and other applications
- [+] very good integrability and automation capacity

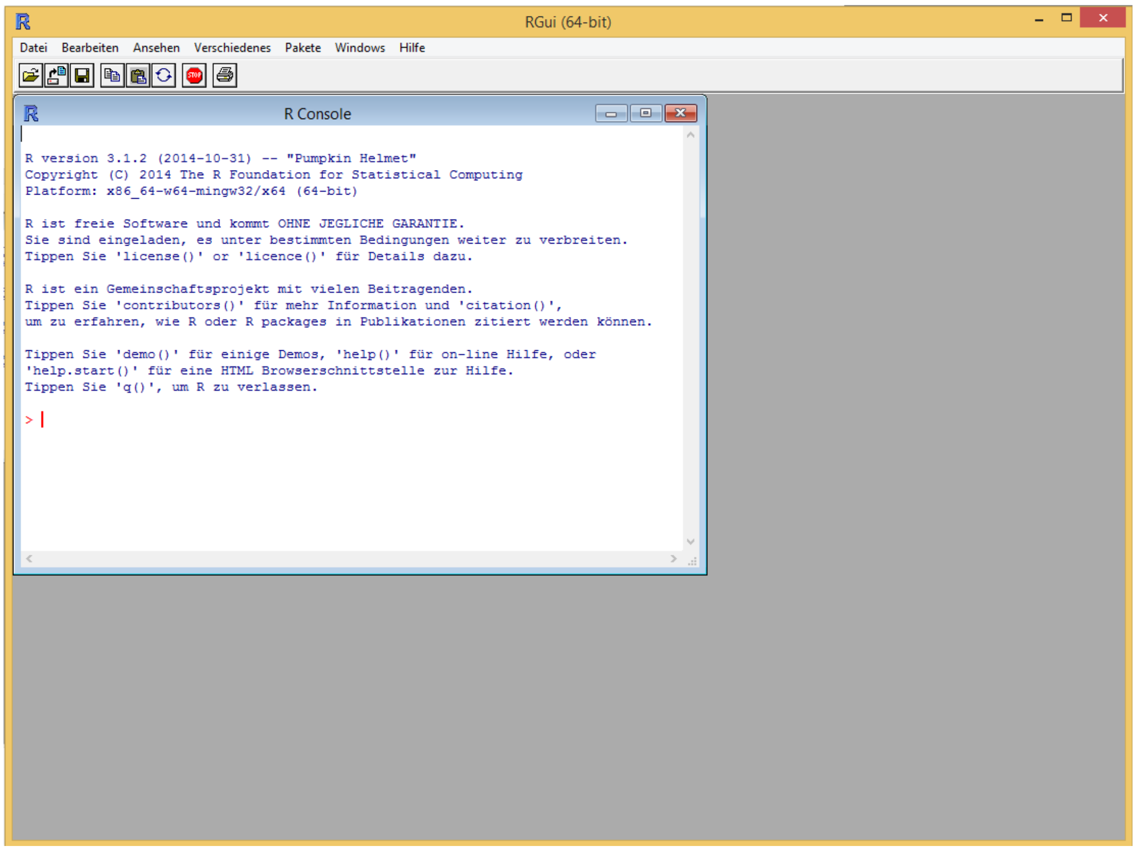
## 1.2 Installation

R sets benchmarks according to its availability across different platforms. There are installers and packages for Windows, OS X and Linux (32 and 64 bit). Furthermore the source code is available for everyone to download, in order to compile the code for diverse other platforms. The setup file for R can be downloaded here: www.r-project.org → CRAN.

Multiple versions of R can be installed in parallel, so the installation of a new R version will not replace the current one. In order to replace the current version, deinstall the old version first and install the new R version subsequently. If you do it the other way round under Windows, the deinstallation will cause the deletion of some registry keys. As a consequence, some programs like RStudio (see section 1.3) may not find R.exe anymore.

Figure 1 illustrates the user interface of R and its console, in which you can type your commands.

So-called *packages* provide most of the functions in R. You can (down)load or update these packages via the menu item "packages".



```
R version 3.1.2 (2014-10-31) -- "Pumpkin Helmet"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R ist freie Software und kommt OHNE JEGLICHE GARANTIE.
Sie sind eingeladen, es unter bestimmten Bedingungen weiter zu verbreiten.
Tippen Sie 'license()' or 'licence()' für Details dazu.

R ist ein Gemeinschaftsprojekt mit vielen Beitragenden.
Tippen Sie 'contributors()' für mehr Information und 'citation()',
um zu erfahren, wie R oder R packages in Publikationen zitiert werden können.

Tippen Sie 'demo()' für einige Demos, 'help()' für on-line Hilfe, oder
'help.start()' für eine HTML Browserschnittstelle zur Hilfe.
Tippen Sie 'q()', um R zu verlassen.

> 
```

**Figure 1** – The R GUI

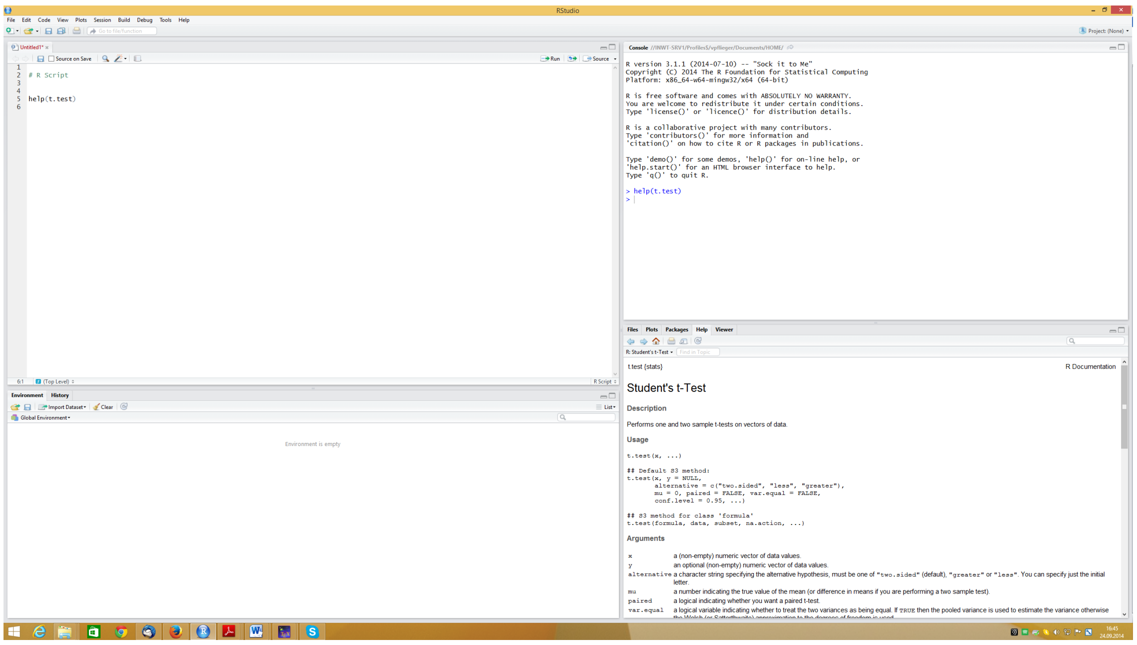
The following packages are available per default: `base`, `grDevice`, `methods`, `utils` (fundamental functions of R), `stats` (fundamental statistical functions), `graphics` (functions to generate graphics), and `datasets` (example datasets). Additional packages are installed but not active automatically. You can get an overview of the installed packages via the buttons "packages" → "load packages..." in R.

## 1.3 Operating Concept and GUIs

The operating concept of R is an advantage and disadvantage simultaneously. Once you have launched R, the only thing you can see is an empty command line. Under Windows, the user interface is quite minimalistic, whereas R provides some additional features under Linux and OS X (i.e., syntax highlighting). Even though with the "R Commander" (https://cran.r-project.org/package=Rcmdr) there exists a rudimentary front-end, it cannot account for R's flexibility and consequently covers only a fraction of its functions. Same holds for the web application "intRo" at http://www.intro-stats.com. The true strength of R lies in the consequent use of its powerful programming language, which supports advanced concepts such as object orientation and enables access to the whole range of R's functions.

In R, everything can be automized and customized down to the least detail. Compared to Excel, you can generate graphics with almost unlimited flexibility, making it easier to create graphics for example in a company's corporate design. This enormous flexibility may, however, increase the time spent dealing with graphics.

You are well-advised to use the comfortable open source development environment *RStudio* (http://rstudio.org/) as an alternative to R's rudimentary text editor. In addition to syntax highlighting, code-folding, and auto-completion, *RStudio* offers features like an integrated object browser, Git-integration, and debugging support (see Figure 2). The *RStudio* client can be installed under Windows, OS X and Linux. Furthermore there is a server version for Linux available that provides the client's full range of functions via every common web browser.



```
# R Script

help(t.test)
```

Console //INWT-SRV1/Profiles$/vpflieger/Documents/HOME/

```
R version 3.1.1 (2014-07-10) -- "Sock it to Me"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> help(t.test)
> 
```

t.test {stats} R Documentation

Student's t-Test

Description

Performs one and two sample t-tests on vectors of data.

Usage

```
t.test(x, ...)

## Default S3 method:
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)

## S3 method for class 'formula'
t.test(formula, data, subset, na.action, ...)
```

Arguments

| | |
|---|---|
| x | a (non-empty) numeric vector of data values. |
| y | an optional (non-empty) numeric vector of data values. |
| alternative | a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter. |
| mu | a number indicating the true value of the mean (or difference in means if you are performing a two sample test). |
| paired | a logical indicating whether you want a paired t-test. |
| var.equal | a logical variable indicating whether to treat the two variances as being equal. If TRUE then the pooled variance is used to estimate the variance otherwise |

16:45
24.09.2014

**Figure 2** – The RStudio interface

Please be aware that a lot of valuable project time can be lost because of encoding problems. This is typically the case when R code is used on Windows as well as Linux. You can find out about the default encoding used by your computer by typing

```
Sys.getlocale()
```

Generally, Windows uses Windows-1252 or CP-1252, whereas Linux uses UTF-8 by default. It is not only the handling of special characters that can cause a conflict. Sometimes the internal sorting mechanism is affected by the encoding... No one likes to run into these problems deliberately! Hence, you are well-advised to save all your R code in UTF-8. In *RStudio*, encoding is set by clicking on "Tools", then choose "Global Options...". On the R General pane you find an option where you can change the default text encoding.

As an alternative to *RStudio*, the *Notepad*++ editor can be extended in order to provide syntax highlighting for R scripts as well. In conjunction with *NppToR* it is possible to send R-code from *Notepad*++ to the console and to execute it. Additionally, you can install the XML file listed below to get the auto-completion feature for Notepad++. All open source features mentioned above can be found under the following URLs:

- **Notepad++:** http://notepad-plus-plus.org/
- **NppToR:** http://sourceforge.net/projects/npptor/
- **R.xml:** http://yihui.name/en/2010/08/auto-completion-in-notepad-for-r-script/

## 1.4 Support

Because of the broad community engagement for R, there are quite a lot of free manuals and tutorials, as well as active forums, blogs and mailing lists. A way to get started with R is to use “swirl – Learn R, in R”, a free collection of interactive R courses for beginners, intermediate and advanced users (http://swirlstats.com/). Another open source platform is OpenIntro (https://www.openintro.org/) which provides different online tutorials in order to learn statistics with R. Two books, both written by Hadley Wickham, that help advanced R users to deepen their knowledge about programming and the building of packages are “Advanced R” (http://adv-r.had.co.nz/) and “R packages” (http://r-pkgs.had.co.nz/).

If you are looking for possibilities to keep up to date with methods and functions, we recommend Quick-R (http://www.statmethods.net/) and R-bloggers (http://www.r-bloggers.com/). Also, the R Journal (http://journal.r-project.org/index.html) appears twice a year as pdf and provides information about changes, new packages and best practices. Several mailinglists (http://www.r-project.org/mail.html) offer individual, quick and competent help, as long as you pay attention to the instructions and the Posting Guide (http://www.r-project.org/posting-guide.html). It is not rare that CORE team members themselves offer their help in these mailinglists. Due to the fact that most of the developers and community members are experienced statisticians, they do not only give advice for programming with R, but help in statistical matters as well.

Taken as a whole, the R support offered by the forums and mailingslists outperforms the support of commercial statistic software with respect to its expertise and reaction time. On the other hand, there is no guarantee for your questions to be answered. This gap is closed by several commercial support providers for R.

If a Google search fails to provide adequate solutions, searching the R project’s website may help out: http://www.r-project.org/ → “Search”. Additionally, the Google search on the R website can be helpful in quite a few cases. If you have forgotten a function’s name of an installed package, you can easily find it via the HTML help: “Help” → “HTML Search”.

However, knowing the name of a function but not the way it works is the most common problem. In this case you can access the help page directly. Enter for example `help(t.test)` or `?t.test` in the command line if you’d like to know how the `t.test()` function works. Furthermore, if you are looking for functions related to the keyword “test”, you can use the `apropos()` command: `apropos("test")`.

Packages often bring their own documentation called *vignette*. You can find it by clicking on the package’s name in the package list or via a search engine search. Vignettes give an overview of what the package does and what its functions serve for.

### Further Reading

- Chambers, J. M. (2008): Software for Data Analysis. New York: Springer. *(Chapters 1 and 2; in-depth introduction to what R is and how it works by one of the developers of R)*
- Chapman, C., & McDonnell Feit, E. (2015): R for Marketing Research and Analytics. Springer. *(Chapters 1 and 2.1)*

# 2 R 101

Now that you are well-informed about the R environment, it is time to start working with it! In this section we will teach you the basics about the different objects in R and how to perform some computing and manipulation. In the grey-colored areas you find code snippets and we invite you to execute them in RStudio on your own. Lines starting with ## contain the R output. Let's start!

At the R prompt we type *expressions*. The <- symbol is the assignment operator.

```
x <- 1
print(x)
```

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

```
x
```

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

```
msg <- "hello"
```

In the first line we assigned the value 1 to the object named x. As you can see, no output is generated by this expression, this is because we didn't ask for any. If we'd like R to show us the content of x we can either explicitly say print(x) or just type the variable name x and execute.

The grammar of the language determines whether an expression is complete or not.

```
# Some R code
x <- # Incomplete expression
+
```

However, if you execute some code and R answers with + it says that whatever expression you wrote, it is not complete, i.e., you forgot a ) or " or the like.

The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored.

## 2.1 Objects

R has five basic or *atomic* classes of objects: character, numeric (real numbers), integer, complex and logical (TRUE/FALSE).

There are also many different data structures, e.g., lists or matrices. But the most basic data structure is the vector.

R objects can also have *attributes*, like:

- names, row names, column names
- dimensions (e.g. matrices, arrays)
- class
- length
- other user-defined attributes/metadata.

Attributes of an object can be accessed using the `attributes()` function. The class of an object can be accessed by the `class()` function.

**Note:** Numbers in `R` are generally treated as numeric objects. If you explicitly want an integer, you need to specify the `L` suffix, for example:

```
x1 <- 3
class(x1)
```

```
## [1] "numeric"
```

```
x2 <- 3L
class(x2)
```

```
## [1] "integer"
```

There is also a special number $Inf$ which represents infinity, e.g., $1/0$. $Inf$ can be used in ordinary calculations, e.g. $1/Inf$ is $0$.

Now let's take a closer look at "vectors".

### 2.1.1 Vectors

The `c()` function in combination with the assignment operator <- can be used to create vectors of objects (the `c` stands for concatenate). Within the `c()` function, you simply separate the objects with commas.

```
# Numeric vector
x <- c(1, 2, 3)
x
```

```
## [1] 1 2 3
```

```
# Character vector
y <- c("Chris", "Anna", "Sophie")
y
```

```
## [1] "Chris"  "Anna"   "Sophie"
```

In R, there is no special scalar vector type: objects that consist of only one element are simply treated as vectors of length one. The elements of a vector need to have the same class, e.g. numbers only. Let's find out the class of x and y:

```
class(x)
```

```
## [1] "numeric"
```

```
class(y)
```

```
## [1] "character"
```

Let's be a real jerk now and combine objects of different class in order to create a vector... guess what happens.

```
z <- c("Zebra", 97.5)
z
```

```
## [1] "Zebra" "97.5"
```

```
class(z)
```

```
## [1] "character"
```

When elements of different classes are mixed in one vector, so-called coercion occurs so that every element in the vector is of the same class. R chooses the appropriate class automatically.

There are sometimes situations in which you need to create empty vectors. This is done by the function vector() (which can be used to create vectors besides the c() function). You can also specify the class of your empty vector, the default is "logical".

```
a <- vector(mode = "numeric")
class(a)
```

```
## [1] "numeric"
```

```
a
```

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

To assign a value to a certain position of a vector (it doesn't matter if the vector is empty or not), you can use rectangular brackets, within which you specify the position. The indices in R begin with 1 (and not with 0 like in some other programming languages).

```
y[1]
```

```
## [1] "Chris"
```

```

a[5] <- 1
a
```

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

```
x[4] <- 4
x
```

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

The length of a vector (the number of elements in it) can be obtained by the function `length()`.

```
length(x)
```

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

### Useful functions to generate vectors

Vectors do not always consist of only a small number of objects. The generation of more complex vectors is supported by the functions `rep()`, `seq()` and the `:` operator.

`rep` stands for replicate and, as the name suggests, the function replicates values as often as you wish. The first argument of `rep()` is the vector or concatenated objects you'd like to repeat, after a comma you specify how many times the objects should be replicated. First, we want the numbers 4, 5 and 6 to be replicated 3 times.

```
rep(c(4, 5, 6), 3)
```

```
## [1] 4 5 6 4 5 6 4 5 6
```

If you don't want to repeat the whole vector a number of times, but each of the elements, use the `each` argument:

```
rep(c(4, 5, 6), each = 3)
```

```
## [1] 4 4 4 5 5 5 6 6 6
```

As mentioned above, we can also give the name of a vector as first argument, so say we want the vector `x` to be replicated 5 times.

```
rep(x, 5)
```

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

The next function we'll learn about is `seq()`, which stands for sequence. So what it does is creating a sequence of numbers from your arguments that can be specified as follows: `seq(from, to, by)`. The argument `by` lets you controll for the increments, the default is set to 1.

```
seq(from = 10, to = 20)
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

```
seq(from = 10, to = 20, by = 5)
```

```
## [1] 10 15 20
```

```
seq(1, 5, 0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Instead of the increment, you might rather define the length of the resulting vector:

```
seq(1, 100, length = 15)
```

```
##  [1]   1.000000   8.071429  15.142857  22.214286  29.285714  36.357143
##  [7]  43.428571  50.500000  57.571429  64.642857  71.714286  78.785714
## [13]  85.857143  92.928571 100.000000
```

A sequence is generated by the `:` operator as well.

```
1:5
```

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

This can be thought of as a short version for `seq(1, 5)`.

**Math Operations**

You can add, multiply, subtract... vectors in `R`. If you take a look at the examples beneath, you'll see that the operations are done element by element, i.e., 1 + 1, 2 + 3, 3 + (-2) and 4 + 0 in the first example.

```
x + c(1, 3, -2, 0)
```

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

```
x - x
```

```
## [1] 0 0 0 0
```

```
x * 3
```

```
## [1]  3  6  9 12
```

```
x * 5 - x / 2
```

```
## [1]  4.5  9.0 13.5 18.0
```

### 2.1.2 Factors

Factors are like vectors, but the set of possible values ("levels") is set. Factors are used to represent categorical data. They can be either ordered or unordered. Internally, factors are stored as "labelled" integer vectors. This can be important in linear modelling because factors are treated specially by the respective functions, e.g., `lm()` and `glm()`. Using factors with labels is better than using integers because factors are self-describing: A variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

```r
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: no yes
```

```r
table(x)
```

```
## x
##  no yes
##   2   3
```

Levels represent the different categories of a factor. The order of the levels can be set using the `level` argument to `factor()`[1].

```r
x <- factor(c("yes", "yes", "no", "yes", "no"), levels = c("yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: yes no
```

Additionally, levels can be relabelled, added and reordered.

```r
# New labels
exam <- c(TRUE, TRUE, FALSE, FALSE, TRUE)
exam <- factor(exam,
               levels = c(TRUE, FALSE),
               labels = c("success", "failure"))
table(exam)
```

```
## exam
## success failure
##       3       2
```

```r
# Reorder levels
exam <- relevel(exam, ref = "failure")
table(exam)
```

```
## exam
## failure success
##       2       3
```

```r
# Add new level
levels(exam) <- c("success", "failure", "re-examination")
exam
```

```
## [1] failure failure success success failure
## Levels: success failure re-examination
```

[1]This can be important in linear modelling because the 1st level is always used as the baseline level.

```r
table(exam)
```

```
## exam
##         success         failure re-examination
##               2               3               0
```

### 2.1.3 Matrices

Matrices are vectors with a dimension attribute. The dimension attribute itself is an integer vector of length 2 (rows and columns as `nrow` and `ncol`). All elements must have the same class. The creation of a matrix can be done via the `matrix()` function.

```r
m <- matrix(nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
```

```r
dim(m)
```

```
## [1] 2 3
```

```r
attributes(m)
```

```
## $dim
## [1] 2 3
```

Matrices are constructed column by column, so entries can be thought of starting in the "upper left" corner and running down the columns.

```r
m <- matrix(1:6, nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

However, the arguments `nrow` and `ncol` do not have to be specified both. The number of elements and the number of rows specify a definite matrix, same applies to the number of elements and the number of columns.

```r
m <- matrix(1:6, nrow = 2)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

You can also create a matrix directly from vectors by adding a dimension attribute.

```
m <- 1:10
m
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
dim(m) <- c(2, 5)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

Another way to create a matrix is by using column-binding or row-binding with `cbind()` and `rbind()`.

```
u <- 1:3
k <- 10:12

cbind(u, k)
```

```
##      u  k
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```


rbind(u, k)
```

```
##   [,1] [,2] [,3]
## u    1    2    3
## k   10   11   12
```

If the vector has not enough elements to fill the whole matrix, R recycles it as many times as needed:

```
matrix(1:3, nrow = 3, ncol = 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    2    2    2
## [3,]    3    3    3
```

Matrix rows and columns can have name attributes, the associated functions are `colnames()` and `rownames()`.

```
colnames(m) <- c("These", "are", "great", "column", "names")
m
```

```
##      These are great column names
## [1,]     1   3     5      7     9
## [2,]     2   4     6      8    10
```

```
rownames(m) <- c("Row_1", "Row_2")
m
```

```
##       These are great column names
## Row_1     1   3     5      7     9
## Row_2     2   4     6      8    10
```

The elements of one-dimensional vectors can also be named by the `names()` function.

```
names(x) <- c("One", "Two", "Three", "Four")
x
```

```
##   One   Two Three  Four  <NA>
##   yes   yes    no   yes    no
## Levels: yes no
```

### Matrix Operations

Matrices can be added, multiplied etc. as well. The following code snippets show you how to do basic matrix operations.

```
# Define two matrices
x <- matrix(1:4, ncol = 2)
y <- matrix(4:1, ncol = 2)

# Matrix multiplication
x %*% y
```

```
##      [,1] [,2]
## [1,]   13    5
## [2,]   20    8
```

```

# Multiplication element by element
x * y
```

```
##      [,1] [,2]
## [1,]    4    6
## [2,]    6    4
```

```

# Addition element by element
x + y
```

```
##      [,1] [,2]
## [1,]    5    5
## [2,]    5    5
```

```

# Transpose
t(x)
```

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

```
# Determinant
det(x)
```

```
## [1] -2
```

### 2.1.4 Data Frames

Data frames are used to store tabular data. Unlike matrices, data frames can store different classes of objects in each column. They are typically used whenever you work with datasets. Data frames also have the attributes `row.names` and `names` (the column names).

```
mydf <- data.frame(Name = c("Max Mendez", "Louisa Lumiere", "Otto Oberman"),
                   Place = c(2, 1, 3),
                   Points = c(138, 204, 99))
mydf
```

```
##             Name Place Points
## 1     Max Mendez     2    138
## 2 Louisa Lumiere     1    204
## 3   Otto Oberman     3     99
```

```
attributes(mydf)
```

```
## $names
## [1] "Name"   "Place"  "Points"
##
## $row.names
## [1] 1 2 3
##
## $class
## [1] "data.frame"
```

### 2.1.5 Lists

Lists are the most general data type in `R`. They are special because they can contain elements of different classes and types.

```
x <- list(1, "a", TRUE)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
```

The elements of a list can be named while creating the list, or afterwards. If we created a list for a new patient at a hypothetical veterinary practice, it could look as follows:

```
y <- list(name = "Bello",
          owner = "Fred Smith",
          animal = "dog",
          breed = "Chihuahua",
          last_vaccination = "12-03-2017")
y
```

```
## $name
## [1] "Bello"
##
## $owner
## [1] "Fred Smith"
##
## $animal
## [1] "dog"
##
## $breed
## [1] "Chihuahua"
##
## $last_vaccination
## [1] "12-03-2017"
```

```
names(x)
```

```
## NULL
```

```
names(y)
```

```
## [1] "name"             "owner"            "animal"
## [4] "breed"            "last_vaccination"
```

A list can be unlisted (i.e., transformed into a vector). In this case the names remain but the structure is torn apart to one dimension.

```
v <- unlist(y)
v
```

```
##             name            owner           animal            breed
##          "Bello"     "Fred Smith"            "dog"      "Chihuahua"
## last_vaccination
##     "12-03-2017"
```

Of course the elements of a list cannot only be vectors of length one, then they would not be that different from a simple vector. Each element can be a vector, matrix, data frame or . . . another list.

```
mylist <- list(vector = u, matrix = m, list = y)
mylist
```

```
## $vector
## [1] 1 2 3
##
## $matrix
##       These are great column names
## Row_1     1   3     5      7     9
## Row_2     2   4     6      8    10
##
## $list
## $list$name
## [1] "Bello"
##
## $list$owner
## [1] "Fred Smith"
##
## $list$animal
## [1] "dog"
##
## $list$breed
## [1] "Chihuahua"
##
## $list$last_vaccination
## [1] "12-03-2017"
```

By the way, data frames are a special case of lists where all elements have the same length.

## 2.2 Subsetting

There are a number of operators that can be used to extract subsets of R objects. We already introduced the square brackets [ ]. They can be used to select either one or more elements. The subset you get by using the single square brackets is always of the same class as the object you take a subset from.

```
x <- c("a", "b", "c", "c", "d", "a")
x
```

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

```
x[2]
```

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

```
x[1:4]
```

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

```
x[x > "a"]
```

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

The next operator are the double square brackets [[ ]], which are used to extract elements of a list or a data frame. They can only be used to extract a single element and the class of the returned object will not necessarily be of the same class as the original object.

```
x <- list(ham = 1:4, cheese = 0.6)
x
```

```
## $ham
## [1] 1 2 3 4
##
## $cheese
## [1] 0.6
```

```
x[1]
```

```
## $ham
## [1] 1 2 3 4
```

```
x[[1]]
```

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

The dollar sign $ is used to extract elements of a list or data frame by name.

```
x$ham
```

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

```r
df <- data.frame(name = c("Roger", "Lola", "Mogli"), count = c(3, 2, 6))
df
```

```
##    name count
## 1 Roger     3
## 2  Lola     2
## 3 Mogli     6
```

```r
df$name
```

```
## [1] Roger Lola  Mogli
## Levels: Lola Mogli Roger
```

```r
df$count
```

```
## [1] 3 2 6
```

The `[[ ]]` operator can be used with computed indices while `$` can only be used with literal names.

```r
x <- list(eggs = 1:4, cheese = 0.6, ham = "hello")
x
```

```
## $eggs
## [1] 1 2 3 4
##
## $cheese
## [1] 0.6
##
## $ham
## [1] "hello"
```

```r
name <- "ham"
x[[name]] ## Computed index for "ham"
```

```
## [1] "hello"
```

```r
x$name ## Element "name" doesn't exist!
```

```
## NULL
```

```r
x$cheese ## Element "cheese" does exist
```

```
## [1] 0.6
```

Another way to subset data frames or matrices is by specifying rows and/or columns within the single rectangular brackets. For example, if you'd like to extract the element in the 3rd row of the 1st column of a data frame called `mydf`, you would write `mydf[3, 1]`. As you can see, the row(s) are defined before the comma, column(s) after the comma.

```r
x <- matrix(1:6, 2, 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
x[1, 2]
```

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

```
x[2, 1]
```

```
## [1] 2
```

```
# Subset entire first row
x[1, ]
```

```
## [1] 1 3 5
```

```
# Subset entire second column
x[, 2]
```

```
## [1] 3 4
```

However, subsetting columns of a data frame just by position can be dangerous: If your data frame changes in the future, you will maybe select different columns than before. That's why it's always safer to use column names for subsetting. The same holds for lists: You should always name all list elements and address them via their name instead of their position.

```
# Dangerous:
df[, 1]
```

```
## [1] Roger Lola  Mogli
## Levels: Lola Mogli Roger
```

```
# Safer:
df$name
```

```
## [1] Roger Lola  Mogli
## Levels: Lola Mogli Roger
```

## 2.3 Missing Values

Missing values are denoted by NA ("not available") or NaN ("not a number") for undefined mathematical operations. NA values have a class as well, so there are integer NA, character NA, etc. A NaN value is also NA but the converse is not true. There are two functions to test if an object contains any missing values.

```
x <- c(3, 8, NA, 22, 0, NA)
is.na(x)
```

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

```
sum(is.na(x))
```

```
## [1] 2
```

```
is.nan(x)
```

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

```
sum(is.nan(x))
```

```
## [1] 0
```

You can see that is.na() returns a logical vector. If an element equals NA, the function returns TRUE and FALSE otherwise. What you can't see, is that R treats a FALSE as 0 and a TRUE as 1, that's why you can calculate how much NA's are in an object by calling sum(is.na(x)).

### 2.3.1 Removing NA values

A common task is to remove missing values from an R object. If we're dealing with a vector like x, one method is to assign the logical vector we created with is.na(x) to a variable and exclude it from x.

```
bad <- is.na(x)
bad
```

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

```
x[!bad]
```

```
## [1]  3  8 22  0
```

The rectangular brackets say that we want a subset from x. The subset is specified within the brackets. The exclamation mark ! negates whatever comes after it. Hence, the last line says that we want x without the elements where bad equals TRUE.

The function complete.cases() can be seen as the contrary to is.na(). You'll see why.

```
complete.cases(x)
```

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

```
good <- complete.cases(x)
x[good]
```

```
## [1]  3  8 22  0
```

Now we will create a character vector `y`, which contains `NA`s, and merge `x` and `y` together as data frame. The function `na.omit()` can be used to directly exclude missing values from an object. But now we have a data frame where there are `NA`s at different positions. Note that `na.omit()` excludes every row containing at least one missing value.

```
y <- c("Mary S.", "Robin W.", NA, "Clara L.", NA, "Thomas S.")
mydf <- data.frame(name = y, orders = x)
mydf
```

```
##        name orders
## 1   Mary S.      3
## 2  Robin W.      8
## 3      <NA>     NA
## 4  Clara L.     22
## 5      <NA>      0
## 6 Thomas S.     NA
```

```
na.omit(mydf)
```

```
##       name orders
## 1  Mary S.      3
## 2 Robin W.      8
## 4 Clara L.     22
```

### 2.3.2 The `NULL` Object

The `NULL` is a special object in `R` and cannot be seen as a missing value. It has defined neutral ("null") behavior and no type and no modifiable properties. Don't confuse the `NULL` with a vector/list of zero length or the number `0`.

Functions and expressions whose value is undefined do often return `NULL`. Also note that `NULL` is one of the reserved words in `R`, hence it can't be used e.g. as variable name. To test if an object is `NULL`, use `is.null()`.

## 2.4 Operators

| Operator(s) | Description |
|---|---|
| $+, -$ | Plus, Minus, unary or binary |
| ! | NOT, unary |
| : | Sequence, binary (in model formulae: interaction) |
| $*, /$ | Multiplication, Division, binary |
| ˆ | Exponentiation, binary |
| $\%\%$ | Modulus, binary |
| $\%/\%$ | Integer divide, binary |
| $\% * \%$ | Matrix product, binary |
| $\%in\%$ | Matching operator, binary (in model formulae: nesting) |
| $<, <=$ | Less than, Less than or equal, binary |
| $>, >=$ | Greater than, Greater than or equal, binary |
| $==$ | Equal to, binary |
| $\&, \&\&$ | AND (vectorized, not vectorized), binary |
| $\|, \|\|$ | OR (vectorized, not vectorized), binary |

In this section you've learned a lot of new stuff. It is totally human if you can't remember every detail. If you ever wonder how a certain function works or what arguments can be specified etc., here's a little help: type a question mark followed by the exact function name (without brackets!) in the `R` prompt. Try for example: `?matrix`

### Further Reading

- Chambers, J. M. (2008): Software for Data Analysis. New York: Springer. *(Chapter 6; in-depth introduction to data types)*
- Chapman, C., & McDonnell Feit, E. (2015): R for Marketing Research and Analytics. Springer. *(Chapter 2.4 and 2.5, appropriate for R newbies)*
- Wickham, H. (2017): Advanced R. *(Chapters 2 and 3)* Available under adv-r.had.co.nz/

### Exercises

Now that you're familar with the basic R operations, let's see what you can remember. Think about the tasks alone and keep in mind that producing an error doesn't mean the end of the world. The solutions can be found at the end of this section.

### Vectors and Matrices

1. Define the following vectors in R:

   - a = sequence from 0 to 8
   - b = (1, 2, 3, 1, 2, 3, 1, 2, 3)
   - c = (1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0)
   - d = (1, 2, 2, 3, 3, 3, 4, 4, 4)

2. What are the dimensions and lengths of a, b, c and d?
3. Define a matrix **A** with the vectors a, b, c and d as columns.
4. What are the dimensions and length of **A**?
5. Change the column names of **A** to "alpha", "beta", "gamma" and "delta".
6. What are the row names of **A**? Change them to an ascending sequence.
7. Create a named vector `namedNumbers` containing the elements (1, 2, 3, 4) with element names ("alpha", "beta", "gamma" and "delta").
8. Extract only the first element of `namedNumbers`, use the indexing-function `[`. First use the position directly, second, use the name and third, use a logical vector, for example:

```
c(TRUE, FALSE, FALSE, FALSE)
```

9. Calculate the mean, standard deviation, min, max and median for `a`.
10. What is the value of `x * y` in the following case? Just think about the answer without using R!

```
y <- c(1, 2)
x <- y
y <- 3
x * y
```

### Lists, Factors and Data Frames

11. Now create a list called `aList` that looks like this:

```
aList
```

```
## $a
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
##
## $b
## [1] "sun"   "earth" "moon"
##
## $c
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
##
## $d
## $d$fruit
## [1] "apple"         "kiwi"          "passion_fruit"
##
## $d$taste_rank
## [1] 3 1 2
```

12. Create a vector `x` such that:

```
x
```

```
##  [1] 2 1 2 3 2 1 2 3 2 1 2 3
```

Then change `x` to a factor with level order 3, 2, 1 and label the levels with “high”, “moderate” and “low”. `x` should look like this when you print it to the console:

```
x
```

```
##  [1] moderate low      moderate high     moderate low      moderate
##  [8] high     moderate low      moderate high
## Levels: high moderate low
```

13. We changed our mind and would like to have “moderate” as the reference level.

14. Create a table for the factor `x`, so that we can see how many times each levels occurs, in a nice way.

15. Now we’d like to create a data frame `Employee` with the name, department and salary for a hypothetical company such that:

```
Employee
```

```
##                Name Department Salary
## 1   Pamela Paragraph      Legal   2800
## 2 Chris Cableclutter         IT   3000
## 3      Norman Number Accounting   2500
```

## Recap

16. Create a data.frame such that

```
data
```

```
##    country  currency population_mio
## 1 Nigeria     Naira            182
## 2  France      Euro             67
## 3   India     Rupee           1276
## 4 Bolivia Boliviano             11
```

17. Subset only the second column from `data` using its name. Assign the values to a new vector called `cur`.

18. Now subset the element of `data` which is in the 3rd row of the 3rd column.

19. Create a matrix from the following elements: a sequence from 1 to 5; one NA value; 7, 6, 5 replicated 3 times; one NA value. It should have 4 rows and have the name `M`.

20. Calculate how many missing values are contained in `M`.

21. Exclude all rows with missing values from the matrix and assign the result to `M` again.

22. Now name the columns of matrix `M` “great”, “matrix”, “column” and “names”.

23. Think first, check your results with `R`. Matrix `M` is defined as before. What is the result of:

```
1 == 2
1 != 2
1 == 2 & 2 == 2
1 == 2 | 2 == 2
"1" == 1
TRUE == 1
FALSE == 0
TRUE != FALSE
is.numeric(c(TRUE, FALSE, TRUE))
is.logical(c(TRUE, FALSE, TRUE))
is.numeric(M)
is.matrix(M)
```

24. Check the class of the object `cur`. You’ll see that `cur` is a factor variable. Print its levels to the console.

25. Change the class of `cur` to “character”. (Hint: Use the function `as.character()` and don’t forget to assign whatever you do to `cur`.)

26. Define a vector `b` such that:

```r
is.na(b)
```

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

```r
is.nan(b)
```

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

## Solutions

1.

```
a <- seq(from = 0, to = 8) # or
a <- 0:8

b <- c(1, 2, 3, 1, 2, 3, 1, 2, 3) # or
b <- rep(c(1, 2, 3), 3)

c <- seq(1, 5, 0.5)

d <- c(1, 2, 2, 3, 3, 3, 4, 4, 4) # or
d <- rep(c(1, 2, 3, 4), times = c(1, 2, 3, 3)) # or
d <- rep(1:4, c(1, 2, 3, 3))
```

2. a, b, c, and d are one-dimensional vectors of length 9.

3.

```
A <- matrix(c(a, b, c, d), ncol = 4)
```

4.

```
dim(A)
```

```
## [1] 9 4
```

```
length(A)
```

```
## [1] 36
```

5.

```
colnames(A) <- c("alpha", "beta", "gamma", "delta")
```

6.

```
rownames(A) <- c(1:nrow(A)) # or
rownames <- seq(1, 9)
```

7.

```
namedNumbers <- c(1:4)
names(namedNumbers) <- c("alpha", "beta", "gamma", "delta")
```

8.

```
namedNumbers[1]
namedNumbers["alpha"]
namedNumbers[c(TRUE, FALSE, FALSE, FALSE)]
```

9.

```
mean(a)
```

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

```
sd(a)
```

```
## [1] 2.738613
```

```
min(a)
```

```
## [1] 0
```

```
max(a)
```

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

```
median(a)
```

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

10.

```
y <- c(1, 2)
x <- y
y <- 3
x * y
```

```
## [1] 3 6
```

### Lists, Factors and Data Frames

11.

```
aList <- list(a = c(seq(1, 5, 0.5)),
              b = c("sun", "earth", "moon"),
              c = matrix(1:20, ncol = 5),
              d = list(fruit = c("apple", "kiwi", "passion_fruit"),
                       taste_rank = c(3, 1, 2)))
```

12.

```r
x <- rep(c(2, 1, 2, 3), 3)
x <- factor(x, levels = c(3, 2, 1), labels = c("high", "moderate", "low"))
```

13.

```r
x <- relevel(x, ref = "moderate")
```

14.

```r
table(x)
```

```
## x
## moderate     high      low
##        6        3        3
```

15.

```r
Employee <- data.frame(Name = c("Pamela Paragraph",
                                "Chris Cableclutter",
                                "Norman Number"),
                       Department = c("Legal", "IT", "Accounting"),
                       Salary = c(2800, 3000, 2500))
```

**Recap**

16.

```r
data <- data.frame(country = c("Nigeria", "France", "India", "Bolivia"),
                   currency = c("Naira", "Euro", "Rupee", "Boliviano"),
                   population_mio = c(182, 67, 1276, 11))
```

17.

```r
cur <- data$currency
```

18.

```r
data[3, 3]
```

19.

```r
M <- matrix(c(1:5, NA, rep(c(7, 6, 5), 3), NA), nrow = 4)
```

20.

```r
sum(is.na(M))
```

```
## [1] 2
```

21.

```r
M <- na.omit(M)
```

22.

```r
colnames(M) <- c("great", "matrix", "column", "names")
```

23.

```r
1 == 2
```

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

```r
1 != 2
```

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

```r
1 == 2 & 2 == 2
```

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

```r
1 == 2 | 2 == 2
```

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

```r
"1" == 1
```

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

```r
TRUE == 1
```

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

```r
FALSE == 0
```

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

```r
TRUE != FALSE
```

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

```r
is.numeric(c(TRUE, FALSE, TRUE))
```

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

```
is.logical(c(TRUE, FALSE, TRUE))
```

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

```
is.numeric(M)
```

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

```
is.matrix(M)
```

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

24.

```
class(cur)
```

```
## [1] "factor"
```

```
levels(cur)
```

```
## [1] "Boliviano" "Euro"      "Naira"     "Rupee"
```

25.

```
cur <- as.character(cur)
class(cur)
```

```
## [1] "character"
```

26.

For example:

```
b <- c(99, 1, NA, 34, NaN, 56, NA)
```

# 3 Data Import and Export

The first hurdle to overcome before starting to clean, manipulate or predict data is the import. In this section you will learn how to import different data files like `.csv` and `.xslx` as well as some skills to import data from a database. Then we'll show you how to check if the import meets your expectations.

## 3.1 Data Import

Since `R` does not provide a data editor suitable for data input, data has to be imported at the beginning of each analysis. In order to read in files of various formats, the following packages can be used[2]:

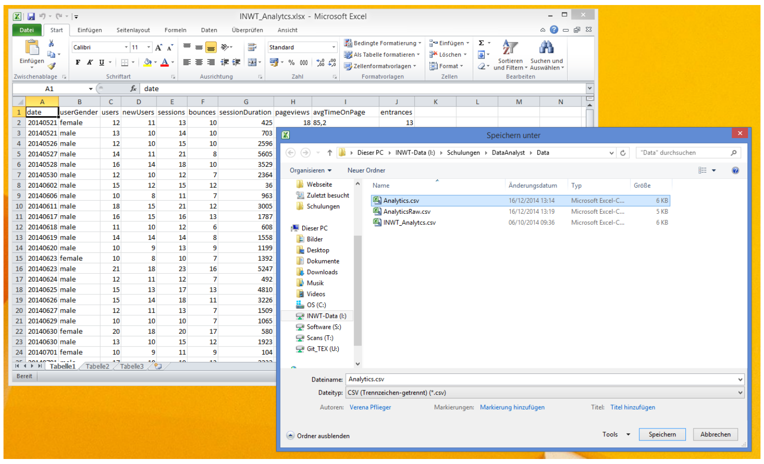
- `readr` (Flat files: csv, csv2, . . . )
- `readxl` (Excel, . . . )
- `haven` (Statistics packages: sas, SPSS, Stata, . . . )

The listed packages have in common that they are faster than the `base` package solution. In addition, character variables are not turned into factors automatically. The datasets are converted into the class “tbl_df”, “tbl”, and “data.frame”, which is a nice feature when working with the `dplyr` package (the most popular package for data manipulation – see Section 5).

When doing smaller analyses, data are usually provided in form of an Excel file. Hence, first we will focus on the import of data from Excel.

### 3.1.1 Import of a csv File

When working with Excel, it is possible to save data in *.csv* format (“comma-separated values”). To do so, open your data with Excel, click “File”, then choose “Save as” and select as type “CSV (Comma delimited)”.



A1 date

| | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | date | userGender | users | newUsers | sessions | bounces | sessionDuration | pageviews | avgTimeOnPage | entrances |
| 2 | 20140521 | female | 12 | 11 | 13 | 10 | 425 | 18 | 85,2 | 13 |
| 3 | 20140521 | male | 13 | 10 | 14 | 10 | 703 | | | |
| 4 | 20140526 | male | 12 | 10 | 15 | 10 | 2596 | | | |
| 5 | 20140527 | male | 14 | 11 | 21 | 8 | 5605 | | | |
| 6 | 20140528 | male | 16 | 14 | 18 | 10 | 3529 | | | |
| 7 | 20140530 | male | 12 | 10 | 12 | 7 | 2364 | | | |
| 8 | 20140602 | male | 15 | 12 | 15 | 12 | 36 | | | |
| 9 | 20140606 | male | 10 | 8 | 11 | 7 | 963 | | | |
| 10 | 20140611 | male | 18 | 15 | 21 | 12 | 3005 | | | |
| 11 | 20140617 | male | 16 | 15 | 16 | 13 | 1787 | | | |
| 12 | 20140618 | male | 11 | 10 | 12 | 6 | 608 | | | |
| 13 | 20140619 | male | 14 | 14 | 14 | 8 | 1558 | | | |
| 14 | 20140620 | male | 10 | 9 | 13 | 9 | 1199 | | | |
| 15 | 20140623 | female | 10 | 8 | 10 | 7 | 1392 | | | |
| 16 | 20140623 | male | 21 | 18 | 23 | 16 | 5247 | | | |
| 17 | 20140624 | male | 12 | 11 | 12 | 7 | 492 | | | |
| 18 | 20140625 | male | 15 | 13 | 17 | 13 | 4810 | | | |
| 19 | 20140626 | male | 15 | 14 | 18 | 11 | 3226 | | | |
| 20 | 20140627 | male | 12 | 11 | 13 | 7 | 1509 | | | |
| 21 | 20140629 | male | 10 | 10 | 10 | 7 | 1065 | | | |
| 22 | 20140630 | female | 20 | 18 | 20 | 17 | 580 | | | |
| 23 | 20140630 | male | 13 | 10 | 15 | 12 | 1923 | | | |
| 24 | 20140701 | female | 10 | 9 | 11 | 9 | 104 | | | |

Dieser PC › INWT-Data (I:) › Schulungen › DataAnalyst › Data

| Name | Änderungsdatum | Typ | Größe |
|---|---|---|---|
| Analytics.csv | 16/12/2014 13:14 | Microsoft Excel-C... | 6 KB |
| AnalyticsRaw.csv | 16/12/2014 13:19 | Microsoft Excel-C... | 5 KB |
| INWT_Analytcs.csv | 06/10/2014 09:36 | Microsoft Excel-C... | 6 KB |

Dateiname: Analytics.csv
Dateityp: CSV (Trennzeichen-getrennt) (*.csv)
Autoren: Verena Pflieger Markierungen: Markierung hinzufügen Titel: Titel hinzufügen

**Figure 3** – Exporting an Excel sheet to .csv format

[2]Information comes mainly from Hadley Wickham's webinar "Getting your data into R" which can be found here: https://github.com/rstudio/webinars/tree/master/11-Getting-Data-into-R

Each row in the newly created file contains the data of a row in the spreadsheet. The values in the rows are separated by a comma:

```
date,userGender,users
20140521,female,12
20140521,male,13
20140526,male,12
```

Now we can continue with reading the dataset using the package `readr`[3]. As Hadley Wickham puts it in the `readr` vignette: "The goal of readr is to provide a fast and friendly way to read tabular data into R" (https://github.com/hadley/readr).[4]

Files like the one above (separated by comma) are common in the US-centric regions and read into R by using the function `read_csv()` from the `readr` package. `read_csv2()` is used for files with semicolon(;)-delimited columns, which are common in European countries. The `read_csv()` functions hold the additional argument `locale`, which allows you to set features like the default time zone, encoding, decimal mark etc.

Please note that the first row of the *.csv* file contains the variable names. The argument `col_names` can be used to indicate whether the first line contains the variable names or not. It is a logical value (`TRUE` or `FALSE`)[5].

But back to work. Let's use the command `read_csv()` to read in the Google Analytics dataset. Therefore we need to install the `readr` package via `install.packages("readr")` and load it.

```
library("readr")
library("dplyr")
```

Now we read in the dataset as mentioned. But first, we save the data path in a new object `pathData` so we don't need to repeat it every time we load data from that folder:

```
pathData <- "PATH_TO_YOUR_DESKTOP/yourDataFolder"
```

Please note the quotation marks as well as the forward slashes (alternatively a twofold backslash can be used "\\").

Now we can paste together the path and the filename with the `paste0()` command. To be able to use the dataset later on, we assign it to an object named `analytics`.

```
analytics <- read_csv(file = paste0(pathData, "Analytics.csv"))
```

Per default, `read_csv()` uses the first row as column names and guesses the column types by looking at their content. If you don't want to leave these decisions to `read_csv()`, you can specify the column types explicitly. Additionally, you can declare columns as factors and preset the levels like we did below with `userGender` and `bouncesHigh`:

```
analytics <- read_csv(file = paste0(pathData, "Analytics.csv"),
                      col_types = cols(
                        date = col_date(format = "%Y%m%d"),
```

[3]More information on `readr` can be found here: http://readr.tidyverse.org/. That's where most of the information provided here comes from.

[4]The function `fread()` from the `data.table` package is even faster. But since `fread()` converts some variable types without being asked to do so, we are going to work with `readr`.

[5]More information on common parameters can be found in the help file via `help(read_csv)`.

```
                        users = col_integer(),
                        newUsers = col_integer(),
                        sessions = col_integer(),
                        bounces = col_integer(),
                        sessionDuration = col_integer(),
                        pageviews = col_integer(),
                        avgTimeOnPage = col_double(),
                        userGender = col_factor(levels = c("female",
                                                           "male")),
                        bouncesHigh = col_factor(levels = c(">=10 bounces",
                                                            "<10 bounces"))
                      ))
```

Note that a similar but less verbose version would be to specify a string containing as many characters as there are columns. “i” stands for integer, “n” for numeric, “c” for character and “l” for logical:

```
read_csv(file = paste0(pathData, "Analytics.csv"),
         col_types = "iiiiiiincc")
```

```
## # A tibble: 102 x 10
##        date users newUsers sessions bounces sessionDuration pageviews
##       <int> <int>    <int>    <int>   <int>           <int>     <int>
##  1 20140521    12       11       13      10             425        18
##  2 20140521    13       10       14      10             703        26
##  3 20140526    12       10       15      10            2596        36
##  4 20140527    14       11       21       8            5605        82
##  5 20140528    16       14       18      10            3529        52
##  6 20140530    12       10       12       7            2364        46
##  7 20140602    15       12       15      12              36        18
##  8 20140606    10        8       11       7             963        31
##  9 20140611    18       15       21      12            3005        39
## 10 20140617    16       15       16      13            1787        22
## # ... with 92 more rows, and 3 more variables: avgTimeOnPage <dbl>,
## #   userGender <chr>, bouncesHigh <chr>
```

If there are no column names, you can specify them via the `col_names` argument. You can even skip the first rows by using the `skip` argument. This can be useful if your data comes with some header containing additional information which is not part of the actual table.

There are even more functions to read tabular data into `R`, which can be found in the `readr()` package help (see the functions starting with `read_`):

```
help(package = readr)
```

### 3.1.2 Import of an xls File

Apart from Excel files that were saved in *.csv* format, data is often provided in the *.xls* or *.xlsx* format. That’s where the `readxl` package comes into use[6]. To sum it up: An advantage

[6]Here again, detailed information can be found under http://readxl.tidyverse.org/.

of this package is that it does not have any external dependencies which makes it easy to install and run on different operating systems. Additionally, non-ASCII characters and datetimes are handled reasonably. Moreover, blank rows and columns are automatically dropped. Here again `tbl_df` class is complementarily added and hence when working with the package `dplyr` nicer printing is ensured.

The function used is called `read_excel()` and its parameters are similar to `read_csv()` or `read_csv2()`. What is specific though, is the additional parameter "sheet" which enables us to pick a specific sheet out of an excel workbook. Also, the `readxl` package does not contain the argument `locale`. You can again specify the column types, but in a slightly different way than for `read_csv()`: You need a character *vector* with as many elements as the number of column. It can contain the strings "skip", "guess", "logical", "numeric", "date", "text" (which means "character") or "list".

First install the packge with `install.packages("readxl")`.

```r
library("readxl")

analytics_xls <- read_excel(path = paste0(pathData, "analytics.xlsx"),
                            sheet = 1,
                            col_types = c("text", "text", rep("numeric", 8)))
```

### 3.1.3 Import of Files from Common Statistical Packages

In order to import files from the most common statistical packages SAS, Stata and SPSS the package `haven` was developed[7]. The following table shows which functions of this package are used for which statistical program:

| **Program** | **Function** |
|---|---|
| `SAS` | `read_sas()` |
| `Stata` | `read_dta()` |
| `SPSS` | `read_por()`, `read_sav()` |

Additionally, there is the `readLines()` function which allows us to read lines of a text file and store them in a character vector. In `R` it is also possible to save the whole workspace (*.RData files) of an `R` session containing all objects loaded in your environment. To load this workspace you can use the function `load()`.

[7]Again, detailed information can be found here: http://haven.tidyverse.org/.

## 3.2 Database

The following section is based mainly on contents of the *r-project.org* website and focuses on RDBMSs (Relational DataBase Management Systems). RDBMSs store data in form of tables (or relations) in the relational model. In a sense they are a subset of DBMS. DBMSs store data in files and are of hierarchical or navigational form. Complementary, information is taken from Hadley Wickham's webinar "Getting your data into `R`" which can be found here: https://www.rstudio.com/resources/webinars/. In addition, RStudio provides an overview of working with databases under http://db.rstudio.com/. This also covers the Connections pane in the `RStudio` IDE, a wizard for database connections[8].

**RDBMSs** are, and we cite the manual "`R` Data Import/Export" from the *r-project.org* website here[9], designed to:

- *Provide fast access to selected parts of large databases.*
- *Summarize and cross-tabulate columns in databases.*
- *Store data in more organized ways than the conventional way.*
- *Enforce security constraints on access to data while providing concurrent access from multiple clients running on multiple hosts.*
- *Act as server to a wide range of clients.*

Some of the common database management systems (DBMS) are `MySQL`, `Oracle`, `PostgreSQL`, `Sybase`, and `Informix`. In order to access a database using the mentioned database management systems with `R` there are basically three "front-end" interfaces that can be used.

**DBI:** One of them is the package *DBI* (DataBase Interface) which can for example be used with the `R`-packages *RMySQL*, *ROracle*, *RPostgreSQL* and *RSQLite*. In this framework commands for data manipulation and evaluation are identical and only the driver of the connection has to be adjusted accordingly. If you use for example a specific provider like *RMySQL* in order to connect to a `MySQL` database, `R` communicates directly with the database.

**ODBC:** Another "front-end" interface is the *ODBC* (Open Database Connectivity) interface. It functions analogously to *DBI* but is accessed in `R` via the package *RODBC*. It is more of a generic provider so a database specific driver has to be installed first. Then, *RODBC* communicates with an `ODBC MySQL` driver which in turn communicates with `MySQL`.

**JDBC:** And finally, for database sources supporting an *JDBC* (Java Database Connectivity) interface the package *RJDBC* serves as interface. Here, we need one step more. *RJDBC* communicates with `Java` which communicates with a `JDBC MySQL` driver which finally communicates with `MySQL`. These additional steps do not matter considerably when working with the database. The code is only marginally slower, the installation, however, is quite more painful.

In the following we want to look at establishing a connection to a `SQLite` database using the *DBI* as front-end package and the package *RSQLite* as back-end.

[8]Working with the Connections pane may appear easier to programming beginners, but using code instead of clicking has the advantage of easy reproducibility.
[9]https://cran.r-project.org/doc/manuals/r-release/R-data.html accessed 2017-02-28.

### 3.2.1 RSQLite

The package *RSQLite* is open source and hence attractive when working with databases. Together with RStudio it will be used in order to access the database, create tables, load data into tables, read out data etc. It comes in very handy since itself contains `SQLite` - hence, no external software has to be installed. Note that the following paragraph is mainly based on the vignette of the package written by Hadley Wickham [10].

Generally, when working with *RStudio* and *RSQLite* the `SQL` language should be known. This is especially helpful when `SQL` commands are directly sent to the database using the `dbExecute`, `dbSendStatement`, `dbSendQuery` or `dbGetQuery` function. Additionally, the *RSQLite* package provides specific functions for executing tasks on the database. Examples are: `dbExistsTable`, `dbReadTable`, `dbWriteTable` etc.

First, we will use `R` in order to set up a `SQLite` database and write the Analytics data to it. Before we are working on the database though, we read our data into `R` using `read_csv`.

```r
# Read in analytics data and setting column types
library(readr)
```

```r
analytics <- read_csv(file = paste0(pathData, "Analytics.csv"),
                      col_types = cols(
                        date = col_date(format = "%Y%m%d"),
                        users = col_integer(),
                        newUsers = col_integer(),
                        sessions = col_integer(),
                        bounces = col_integer(),
                        sessionDuration = col_integer(),
                        pageviews = col_integer(),
                        avgTimeOnPage = col_double(),
                        userGender = col_factor(levels = c("female",
                                                           "male")),
                        bouncesHigh = col_factor(levels = c(">=10 bounces",
                                                            "<10 bounces"))
                      ))
```

```r
# Checking the data structure
str(analytics)
```

Now we use the *DBI* package in order to initialize a temporary database called `dataScience` on-disk. Once we disconnect from it, the database will automatically be deleted.

Note that we create this temporary database for demonstration purposes. Connecting to a remote database works the same way except that you need to provide some credentials to authorize yourself. In this case it makes sense to store the credentials in a list and access them when necessary:

```r
# Storing database credentials in a list
sqlCredentials <- list(username = "dataScientist",
                       password = "!dataScience@INWT",
```

[10] https://cran.r-project.org/web/packages/RSQLite/vignettes/RSQLite.html accessed 2017-05-12.

```
                       dbname = "dataScience",
                       table = "Analytics",
                       host = "192.168.188.7",
                       port = 3306)

# Step 1: Opening the connection
con <- dbConnect(RSQLite::SQLite(),
                 username = sqlCredentials$username,
                 password = sqlCredentials$password,
                 dbname = sqlCredentials$dbname,
                 host = sqlCredentials$host,
                 port = sqlCredentials$port)

# Step 2: Checking on tables, Writing the data
dbWriteTable(con, sqlCredentials$table, analytics,
             append = TRUE, overwrite = FALSE, row.names=FALSE)

# Step 3: Closing the connection between the DBMS and RStudio
dbDisconnect(con)
```

Anyways, we continue to create our temporary database on-disk where we do not need credentials to work with.

```
library(DBI)
# Creating database on-disk
dataScience <- dbConnect(RSQLite::SQLite(), "")
```

Next, we use `dbExecute` in order to initialize our table. `dbExecute` is used if you do not expect a result to be returned. This can be initializing, updating, inserting into or deleting a table. Internally, a call to `dbExecute` translates to `dbSendStatement`, `dbGetRowsAffected` and `dbClearResult`. Look at the help files to get further information.

During set-up we have to be careful since the definition of the variable types is essential. If they are misspecified the imported variables are converted accordingly and may become illegible. So let's create our table and check the results.

```
# Creating the table
sqlStatement <- "CREATE TABLE Analytics
                 (id INTEGER PRIMARY KEY, -- Autoincrement
                 date TEXT,
                 userGender VARCHAR(10),
                 users INT,
                 newUsers INT,
                 sessions INT,
                 bounces INT,
                 sessionDuration INT,
                 pageviews INT,
                 avgTimeOnPage DOUBLE,
                 bouncesHigh VARCHAR(15))"
```

```r
dbExecute(dataScience, sqlStatement)
```

```
## [1] 0
```

```r
# Checking the result
dbListTables(dataScience)
```

```
## [1] "Analytics"
```

```r
dbListFields(dataScience, "Analytics")
```

```
##  [1] "id"              "date"            "userGender"
##  [4] "users"           "newUsers"        "sessions"
##  [7] "bounces"         "sessionDuration" "pageviews"
## [10] "avgTimeOnPage"   "bouncesHigh"
```

```r
dbReadTable(dataScience, "Analytics")
```

```
##  [1] id              date            userGender      users
##  [5] newUsers        sessions        bounces         sessionDuration
##  [9] pageviews       avgTimeOnPage   bouncesHigh
## <0 rows> (or 0-length row.names)
```

Note that we could have used `dbGetQuery` or `dbSendQuery` instead. `dbSendQuery` (which allows us to send a `SQL` query to the database) is used in combination with `fetch` (in order to get the results into `R`) and `dbClearResult` (in order to clear the results from the connection). As alternative to `dbGetQuery` it is rather common but bears the risk of forgetting to fetch and clear the results and hence provoking the following error:

```
Error in .local(conn, statement, ...) :   connection with pending rows, close
resultSet before continuing
```

It has, however, the advantage that you can set the amount of records to be imported into `R` with the parameter `n` in the `fetch()`-function (-1 implies that all records are imported).

Anyways, using `dbExecute` worked. We created the structure of the table which needs to be filled with data. We can write data to the table from within *RStudio* using the function `dbWriteTable()`.

```r
# Write to database
dbWriteTable(dataScience,
             "Analytics",
             analytics,
             append = TRUE,
             overwrite = FALSE,
             row.names = FALSE)
# Check the result
dbReadTable(dataScience,
            "Analytics") %>% head(n = 3)
```

```
##   id    date userGender users newUsers sessions bounces sessionDuration
## 1  1 16211.0     female    12       11       13      10             425
## 2  2 16211.0       male    13       10       14      10             703
## 3  3 16216.0       male    12       10       15      10            2596
##   pageviews avgTimeOnPage  bouncesHigh
## 1        18        85.200 >=10 bounces
## 2        26        58.500 >=10 bounces
## 3        36       123.619 >=10 bounces
```

Besides `dbReadTable` there are other ways of reading data from a database. One way is to use a `SQL` query executed via `dbGetQuery`. First we simply read in the whole *Analytics* dataset.

```
# Reading data with dbGetQuery()

sqlQuery <- "SELECT * FROM Analytics"
datSQL <- dbGetQuery(dataScience, sqlQuery)
head(datSQL, n = 3)
```

```
##   id    date userGender users newUsers sessions bounces sessionDuration
## 1  1 16211.0     female    12       11       13      10             425
## 2  2 16211.0       male    13       10       14      10             703
## 3  3 16216.0       male    12       10       15      10            2596
##   pageviews avgTimeOnPage  bouncesHigh
## 1        18        85.200 >=10 bounces
## 2        26        58.500 >=10 bounces
## 3        36       123.619 >=10 bounces
```

Now assume that we do not want to read in all data but select single rows which meet a certain criteria.

```
# Reading lines of female users
sqlQuery <- "SELECT * FROM Analytics WHERE `userGender` = 'female'"
datSQL <- dbGetQuery(dataScience, sqlQuery)
datSQL %>% head(n = 3)
```

```
##   id    date userGender users newUsers sessions bounces sessionDuration
## 1  1 16211.0     female    12       11       13      10             425
## 2 14 16244.0     female    10        8       10       7            1392
## 3 21 16251.0     female    20       18       20      17             580
##   pageviews avgTimeOnPage  bouncesHigh
## 1        18         85.20 >=10 bounces
## 2        15        278.00  <10 bounces
## 3        24        144.75 >=10 bounces
```

```


# Reading lines where avgTimeOnPage is below 25
sqlQuery <- "SELECT * FROM Analytics WHERE `avgTimeOnPage` < 25"
```

```
datSQL <- dbGetQuery(dataScience, sqlQuery)
datSQL %>% head(n = 3)
```

```
##   id    date userGender users newUsers sessions bounces sessionDuration
## 1  7 16223.0       male    15       12       15      12              36
## 2 33 16261.0       male    11        7       11       6             139
## 3 57 16301.0       male    18       14       21      16             293
##   pageviews avgTimeOnPage  bouncesHigh
## 1        18      12.00000 >=10 bounces
## 2        18      20.00000  <10 bounces
## 3        34      22.61538 >=10 bounces
```

### 3.2.2 Parameterized Queries

When automatically updating tables, selecting or deleting variables from databases or the like, there are two features you should be aware of:

- speeding up query execution
- avoiding SQL injections.

While the concept of speeding up query execution is probably self-explaining, you might wonder what SQL injections are.

An "SQL injection is an attack against your code which uses SQL queries. Malicious query parameter values are passed in order to modify and execute a query" (Mateusz Zoltak 2014[11]).

A common example is the `SQL` query below. Consider the following function to find records of a certain student in a database. Note that it uses the `paste0` function:

```
find_student <- function(db, name){
  sql <- paste0("SELECT * FROM Students",
 "WHERE (name = '", name, "'); ")
  dbGetQuery(db, sql)
}
```

Now imagine we need to find a student called "Robert'); DROP TABLE Students; –" (ok, that's of course a silly example but it states the point). Then our `SQL` statement looks like this:

```
SELECT * FROM Students
  WHERE (name = 'Robert');
  DROP TABLE Students; --');
```

The problem here is that thanks to this strange name, the `SQL` query first selects the records from students called Robert. And then, it drops the table with all the students' records (see figure 4). So even though, by using `paste()` or `sprintf()`, it is possible to build parameterized queries, this approach does not protect your code against so-called "SQL Injections".

[11] https://cran.r-project.org/web/packages/RODBCext/vignettes/Parameterized_SQL_queries.html accessed on 2017-05-29

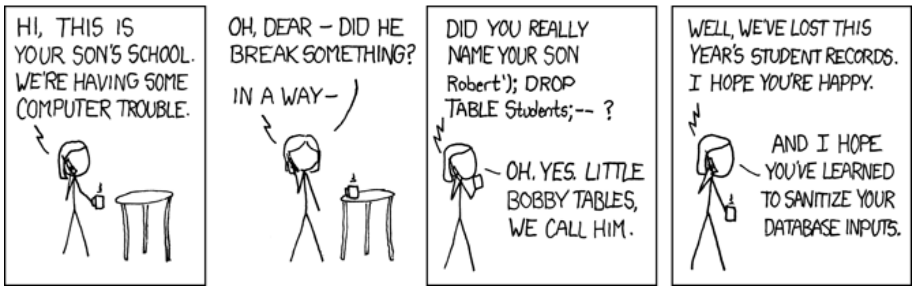

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

**Figure 4** – A comic illustrating SQL injection (Source: http://xkcd.com/327/)

In order to avoid SQL injections, you can use so-called parameterized queries. Parameterized queries enable you to seperate `SQL` statements stored as string from predefined parameters given to these statements. This way you can avoid creating your query by `paste` or `sprintf`. For the example introduced above, a paramterized query looks like the following:

```
find_student <- function(db, name){
  sql <- "SELECT * FROM Students WHERE (name = :student);"
  dbGetQuery(db, sql, params = list(student = name))
}
```

With this structure of a query, you are safe against SQL injections.

Going back to our *Analytics* dataset above and the selection of female users only, we want to dynamically set the the selection criteria using a parameterized query:

```
# Reading lines of female users
criteria <- 'female'
sqlQuery <- "SELECT * FROM Analytics WHERE `userGender` = :x"
datSQL <- dbGetQuery(dataScience,
                     sqlQuery,
                     params = list(x = criteria))
nrow(datSQL)
```

```
## [1] 27
```

```
datSQL %>% head(n = 3)
```

```
##   id    date userGender users newUsers sessions bounces sessionDuration
## 1  1 16211.0     female    12       11       13      10             425
## 2 14 16244.0     female    10        8       10       7            1392
## 3 21 16251.0     female    20       18       20      17             580
##   pageviews avgTimeOnPage  bouncesHigh
## 1        18         85.20 >=10 bounces
## 2        15        278.00  <10 bounces
## 3        24        144.75 >=10 bounces
```

You can also define multiple criteria. In the following query we select only female users with at least 13 bounces, and male users with at least 14 bounces.

```
genders <- c('female', 'male')
nBounces <- c(13, 14)

sqlQuery <- "SELECT * FROM Analytics
              WHERE `userGender` = :x AND `bounces` >= :y"
datSQL <- dbGetQuery(dataScience,
                     sqlQuery,
                     params = list(x = genders,
                                   y = nBounces))

# Check result using table
table(datSQL$userGender, datSQL$bounces)
```

```
##
##          13 14 15 16 17 19
##   female  1  2  0  0  1  0
##   male    0  3  5  2  1  2
```

Note that all criteria in the `params` list need to have the same length. Therefore, to set the same minimum number of bounces for both genders, repeat the number as often as you need it (which will be twice in this case):

```
genders <- c('female', 'male')
nBounces <- rep(13, length(genders))

sqlQuery <- "SELECT * FROM Analytics
              WHERE `userGender` = :x AND `bounces` >= :y"
datSQL <- dbGetQuery(dataScience,
                     sqlQuery,
                     params = list(x = genders,
                                   y = nBounces))

# Check result using table
table(datSQL$userGender, datSQL$bounces)
```

```
##
##          13 14 15 16 17 19
##   female  1  2  0  0  1  0
##   male    7  3  5  2  1  2
```

### 3.2.3 Closing the Connection

Once we are finished working with the database, we can close the connection. Thereby, we remove the database from disk.

```
# When disconnecting database vanishes
dbDisconnect(dataScience)
```

## 3.2.4 Pitfalls

There are few problems that can be encountered when working with SQL and R:

- R is not capable of handling very large integers. So when you are working with large integers it is possible that R internally rounds the number without giving a warning or anything else.

One example:

```
options(digits = 20)
1182032907493978952
```

```
## [1] 1182032907493978880
```

It is possible to code this integers as characters. Then the true number is kept:

```
"1182032907493978952"
```

```
## [1] "1182032907493978952"
```

When working with a database this can lead to confusion since SQL for example holds a special type of variable for big numbers called `BIGINT(20)`. So when you import data into R it is possible that numbers are rounded automatically.

- Encoding can be a problem when working with databases and R. Sometimes it is hard to figure out which encoding which data is stored in. This can be especially cumbersome when working with several databases. When working with *RSQLite*, however, default encoding is `utf8`.

### Very large integers in R

First we create test data that allows us to take a closer look at the problem just mentioned. The first two columns contain really big integers but notice that they should differ within an observation (let's assume they serve as some kind of identifiers). The third column holds a dummy variable, indicating e.g. whether a person clicked on a certain banner on a webpage.

```
# First we create test data
testData <- data.frame(ckid = c(1321990948825077760,
                                1186406680141765888,
                                1123118034563635328,
                                1171727685605399808,
                                1321991018316306432,
                                1182032907493978880),
                       ckidInt = c(1321990948825077845,
                                   1186406680141765963,
                                   1123118034563635286,
                                   1171727685605399875,
                                   1321991018316306517,
                                   1182032907493978952),
                       clickin = c(1, 0, 1, 0, 1, 1))
```

If we check, whether or not we entered the numbers in the right way, we encounter the first peculiarity: The first two columns `ckid` and `ckidInt` hold the same numbers. This is, since R internally rounds big integers.

```
options(digits = 20)
testData
```

```
##                   ckid             ckidInt  clickin
## 1 1321990948825077760 1321990948825077760        1
## 2 1186406680141765888 1186406680141765888        0
## 3 1123118034563635328 1123118034563635328        1
## 4 1171727685605399808 1171727685605399808        0
## 5 1321991018316306432 1321991018316306432        1
## 6 1182032907493978880 1182032907493978880        1
```

Since, R has rounded the numbers before we wrote them to the database we have no chance of reading them back into R in their original format. A workaround here is to load the data directly from a data file on disk into the database, such that R does not get the chance of rounding. Then, we can read the data from the database as characters and hence preserve the initial information. Note that the issue with the big integers is only an issue if we really need the according accuracy. This can e.g. be the case if we are concerned with an identifier in form of big numbers. If we are talking about numbers that really attribute the value of the big integer to an observation, R's rounding behavior should not matter since we probably won't need that kind of accuracy.

Help for dealing with `RSQLite` can be found on the package's webpage at https://CRAN.R-project.org/package=RSQLite. The package's vignette contains examples for the basic functions of `RSQLite`. The reference manual contains extensive descriptions on various aspects of the package. `SQL` commands can be found via http://www.w3schools.com/sql/default.asp, just to name one source of many.

## 3.3 Checking the Import

In any way, no matter how you imported the data, up in the right corner of *RStudio*, in the *Environment* tab, the `analytics` table should occur in your workspace. In order to control the import you can open the data in the editor of `R` using the `fix()` function. Yet, editing data with this crude editor is not really recommended since changes are not tracked and thus not reproducible.

```r
fix(analytics)
```

Alternatively, in order to get an overview of your data and its structure, important commands are for example `summary()`, `str()`, `names()` and `head()`. And when working with `dplyr`, it is possible to simply enter the name of the dataset in order to catch a first glimpse of the data.

In order to avoid problems with data quality in later stages of the analysis, there are ways to check the data after they have been imported. There are for example packages like `assertr` and `ensurer` which contain some functions to control data and the output of computations[12]. You are well-advised to use them. Not only do you ensure that the data you imported is of good quality. Also, by writing adequate tests for your analysis, the analysis itself becomes more structured and clear. Additionally, it lets you reconsider the collection process of your data and makes you reevaluate the amount and kind of data you need.

To find out what to check, it helps to keep the actual meaning of the data in mind. This helps you to find tests for the plausibility of your values. For example, the date someone entered school cannot lie before the date of birth, or the number of driven kilometers of a rented car cannot exceed a specific value depending on the duration. In addition, plots can be very useful to check the data but this is beyond the scope of this course.

[12]Note that there are plenty of packages that have been built in order to control data input, output and the like.

### 3.3.1 assertr

“Essentially, `assertr` provides a suite of functions designed to verify assumptions about data early in an analysis pipeline.” (Toni Fischetti)

In the following we show some examples taken from the package’s vignette written bei Toni Fischetti[13].

He works with `R`’s built-in car dataset `mtcars` which contains information about 32 car models on 11 variables[14].

Type `mtcars` to look at the data and `help(mtcars)` to see its description. `assertr` contains for example the `verify()` function that lets you specify requirements concerning your data. In the following, before data is grouped and the mean calculated, it is checked whether all entries in the variable `mpg` (miles per gallon) are greater or equal to 0.

```
library("dplyr")
library("assertr")

mtcars %>%
  verify(mpg >= 0) %>%
  group_by(cyl) %>%
  summarise(avgMpg = mean(mpg))
```

If the requirement is met, i.e., if all entries in the variable `mpg` are greater than or equal to 0, the mean is returned. If not, an error message will be returned. You are probably wondering about this funny code snippet `%>%`. This is the so-called pipe operator. It basically takes the object the expression started with, in this case the dataset `mtcars`, and executes the next part of the code on it. Thus, from the `mtcars` dataset the variable `mpg` is taken and checked whether or not it is greater or equal to 0. If this is true, the process continues to the next line and the values are grouped by the levels of `cycl`. This object again is taken and the pipe operator “takes” it to the next operation: it is summarised and a new variable is generated filled with the average values.

In order to check more complex requirements on multiple variables, the `assert()` function can be used. It checks one of the following predicates:

- `not_na` (not missing),
- `within_bounds` (within a certain interval),
- `in_set` e.g. for misspelling,

and other custom functions.

So the identical test made above for the miles per gallon being greater or equal to zero can be written with `assert()` as follows:

```
mtcars %>%
  assert(within_bounds(0, Inf), mpg) %>%
  group_by(cyl) %>%
  summarise(avgMpg = mean(mpg))
```

[13] https://cran.r-project.org/web/packages/assertr/vignettes/assertr.html accessed on 2017-02-28
[14] Henderson and Velleman (1981), Building multiple regression models interactively. Biometrics, 37, 391411.

Whereas `verify()` can check the structure of data which `assert()` cannot, `verify()` cannot tell which value exactely violates the requirement. This in turn is something that `assert()` does fairly well (Note the (.) as placeholder for the value itself.).

```
# Checking structure of data with verify:
dat %>% verify(nrow(.) > 100)

# Detailed error with assert:
## Error:
## Vector 'mpg' violates assertion 'within_bounds' 1 time
## (value [-18.7] at index 5)
```

Additionally it is possible to dynamically generate the predicates for `assert()` by using `insist()` (in this case the requirement is that the miles per gallon variable does not vary more than 3 standard deviations):

```
mtcars %>%
  insist(within_n_sds(3), mpg) %>%
  group_by(cyl) %>%
  summarise(avgMpg = mean(mpg))
```

Through the piping operator %>% it is possible to chain several tests together. Note, however, that when the first test fails, testing is interrupted. So even if you manipulate the data such that all the tests should fail, you only get an error for the first test in line. In order to test for several requirements and get an error for each, the `ensurer` package does a better job (see Section 3.3.2).

```
checkMe <- . %>%
  verify(nrow(.) > 10) %>%
  verify(mpg > 0) %>%
  insist(within_n_sds(4), mpg) %>%
  assert(in_set(0, 1), am, vs)

mtcars %>%
  checkMe %>%
  group_by(cyl) %>%
  summarise(avgMpg = mean(mpg))
```

### 3.3.2 ensurer

"The ensurer package has one aim: to make it as simple as possible to ensure expected/needed/desired properties of your data, and to take proper action upon failure to comply." This is how Stefan Holst puts it in the package's vignette[15]. The comments below are taken from the vignette.

There are basically three functions: `ensure_that()` (shorthand `ensure()`), `ensures_that()` (shorthand `ensures()`) and `check_that()`. `ensure_that()` lets you formulate some requirements on objects you are working with. In the following there is a matrix `M` defined but before it is assigned to `theMatrix`, it is checked to be square and composed of numeric values. If this is not the case, an error is printed. Again, (.) serves as placeholder for the value itself.

```r
library(ensurer)

M <- matrix(1:9, nrow = 3)

theMatrix <- M %>%
  ensure_that(is.numeric(.),
              NCOL(.) == NROW(.))
```

`ensures_that()` lets you define reusable requirements such that you don't have to type them over and over again.

```r
matrixIsSquare <- ensures_that(NROW(.) == NCOL(.))
all_positive <- ensures_that(all(. > 0))

matrix(runif(16), 4, 4) %>%
  ensure(+matrixIsSquare, +all_positive)
```

Note that conditions in one `ensure()` call are all checked. If there are several `ensure_that()` calls connected by the piping operator, the conditions are checked subsequently. If one of the first fails, the checking stops. Hence, in order to have all requirements checked, you should place them all in one `ensure()` call.

If you are only interested in `TRUE` or `FALSE`, you can use the function `check_that()`.

```r
mySequence <- 1:5
check_that(mySequence, is.numeric)
```

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

Now see what happens if you redefine `mySequence <- c("test", 1, 4)...`

Note that there are additional options like adding messages, ensuring that the function returns a value, etc: the `fail_with()` argument controls the output when values are not as expected. The default is to raise an error.

The argument `err_desc()` lets you specify an additional error message.

Another great feature is the possibility to build on the `ensure_that()` function and check whether the data match a prespecified template, defining column names, column classes

[15] https://cran.r-project.org/web/packages/ensurer/vignettes/ensurer.html accessed on 2017-05-25

and modes. The following is rather advanced R code, so if you feel too confused you should just come back later...

```r
# Template Definition
carsTemplate <- data.frame(mpg = numeric(0),
                           cyl = numeric(0),
                           disp = numeric(0),
                           hp = numeric(0),
                           drat = numeric(0),
                           wt = numeric(0),
                           qsec = numeric(0),
                           vs = numeric(0),
                           am = numeric(0),
                           gear = numeric(0),
                           carb = numeric(0))
```

```r
# Function to check data
ensureAsTemplate <- function(x, tpl) {
  ensure_that(x,
              is.data.frame(.),
              all(.[, 1] > 10),
              all(.[, 2] > 0),
              identical(class(.), class(tpl)),
              identical(sapply(., class), sapply(tpl, class)),
              identical(sapply(., levels), sapply(tpl, levels)))
}

# Calling on Function
ensureAsTemplate(mtcars, tpl = carsTemplate)
```

## 3.4 Data Export

There are several ways and functions to save files of various formats, for example:

- `write_csv()` from the `readr` package writes to a .csv file.
- `write.table()` prints a matrix or a data frame to a file or connection.
- `save.image()` saves the current workspace (*.RData files).
- `save()` saves specific `R` objects to a file.
- `write()` is used to write data (usually matrices) to a file.
- In order to save various other formats, the `foreign` package can be used.

The most important are probably `write_csv()` and `save()`. We will look at them in greater detail. For the remaining functions, see the respective help files for more information.

### 3.4.1 `write_csv()`

*.csv* files are an easy way to exchange data with others.

`write_csv()` is the counterpart of `read_csv()` from the `readr` package by Hadley Wickham. It has some advantages over `write.csv()` from the `utils` package, e.g., it is faster and does not write row names (which you hardly ever need when working with data).

With its arguments, you can decide how `NA` values are represented and if you want to overwrite existing files or append the new rows to the existing data.

If you need more freedom of choice, you can use the related function `write_delim` and choose a delimiter other than the comma.

### 3.4.2 `save()`

Sometimes you want to save an `R` object for later use in `R`. For example, you may have computed a statistical model and want to create plots from it at a later point. Or you have prepared a dataset that you will need for further analysis. Then you can just save the object as an *.RData* file with `save()`:

```r
# Rename speed variable in the cars dataset
names(cars)[names(cars) == "speed"] <- "speedMph"
# Compute speed in kmh
cars$speedKmh <- cars$speedMph * 1.6093

# Save the modified dataset
save(cars, file = paste0(pathData, "cars_prepared.RData"))
```

You can also save multiple objects in one *.RData* file. When you `load` the file, all the objects are loaded into your workspace. They keep their original names, so you don't need to assign a new name like with `read_csv()`.

`save.image()` is the big brother of `save()`: It saves all objects in your workspace into a file. But be careful: If there are multiple big objects (e.g., several large data frames), the file can become quite large and it can take a long time to load it later on. So it's better to keep control and select the objects to save using `save()`.

## Further Reading

- Chapman, C., & McDonnell Feit, E. (2015): R for Marketing Research and Analytics. Springer. *(Chapter 2.6, good for R newbies, uses base R)*
- Fischetti, T. (2017): Assertive R Programming with assertr. Available under cran.r-project.org/web/packages/assertr/vignettes/assertr.html
- Holst Milton Bache, S. (2015): The ensurer package. Available under cran.r-project.org/web/packages/ensurer/vignettes/ensurer.html
- Wickham, H., & Grolemund, G. (2017). R for Data Science. Available under r4ds.had.co.nz/
- Introduction to readr. `readr` vignette, available under cran.r-project.org/web/packages/readr/vignettes/readr.html

## Exercises

### Import and Export of a csv file

The following exercise is intended to practice writing and reading a dataset.

1. Create a new object called `irisData` and assign the `iris` dataset to it. The `iris` dataset is contained in the package *datasets* which is automatically loaded.
2. Now go ahead and write the `irisData` as *.csv* file to your `Data` folder.
3. Finally, reimport the dataset using `read_csv`. Specify the column types and factor levels during import. If you like, go ahead and see what happens, if you misspecify the factor levels.

### Database

The next exercises aim at making you familiar with working with a dataset that is stored in a database.

4. Keep working with the *Analytics* dataset.
    - Create a database on-disk and define a connection called `myDB`.
    - Set up the table for the *Analytics* data and check your result.
    - Write the data to the table and check your result.
5. Now use parameterized queries to...
    - ...select those observations concerning male users only.
    - ...select those observations that contain 13, 20 and 21 sessions. Work with `dbBind`.
6. Tidy up:
    - Remove the *Analytics* table and check your results.
    - Close the connection.

### Checking the data

Once the data is imported you can check whether it meets certain conditions using *assertr* and *ensurer*.

7. Reconsider the following requirements and then check whether or not the dataset `mtcars` meets them.
    - The number of rows should not fall below 11.
    - The variable `mpg` only contains positive entries.
    - The variable `mpg` stays inside a range of two standard deviations.
    - The variables `vs` and `am` contain only 0s and 1s.
8. Now manipulate the dataset in such a way that all of the requirements fail. Since you are not familiar with *dplyr* yet, you might want to come back to this exercise later on. Or you just try and see what you can do...

9. Now use `ensurer`'s functionality of checking several conditions simultaneously. Start with the following two requirements:

   - The number of rows should not fall below 11 and
   - the miles per gallon should not contain negative values. Check the original `mtcars` dataset as well as the manipulated one from above.

10. If that worked, try to implement a third requirement that the values of the variable `mpg` should not lie outside two standard deviations on each side of the mean (here it makes sense to use two separate statements so that we know whether the according value was too high or too low).

## Solutions

### Import and Export of a csv file

1. We can directly access the dataset typing `iris`. Hence, we use the assignment operator to name it `irisData`.

```
irisData <- iris
```

2. In order to save the dataset to our `Data` folder, we can use the default settings from `write_csv()`.

```
write_csv(irisData,
          path = paste0(pathData, "irisData.csv"))
```

3. The four columns describing length and width of the irises' greens should be read as doubles. Specifying the factor levels can prevent some surprising errors as the analysis goes on.

```
read_csv(file = paste0(pathData, "irisData.csv"),
         col_types = cols (
           Sepal.Length = col_double(),
           Sepal.Width = col_double(),
           Petal.Length = col_double(),
           Petal.Width = col_double(),
           Species = col_factor(levels = c("setosa", "virginica", "versicolor"))
           ))
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1         5.10        3.50         1.40       0.200 setosa
##  2         4.90        3.00         1.40       0.200 setosa
##  3         4.70        3.20         1.30       0.200 setosa
##  4         4.60        3.10         1.50       0.200 setosa
##  5         5.00        3.60         1.40       0.200 setosa
##  6         5.40        3.90         1.70       0.400 setosa
##  7         4.60        3.40         1.40       0.300 setosa
##  8         5.00        3.40         1.50       0.200 setosa
##  9         4.40        2.90         1.40       0.200 setosa
## 10         4.90        3.10         1.50       0.100 setosa
## # ... with 140 more rows
```

### Database

4. Creating a database called `myDB` on-disk:

```
# Creating database on-disk
myDB <- dbConnect(RSQLite::SQLite(), "")
```

Setting up the table for the *Analytics* data using `dbExecute` and checking the result:

```
# Creating the table
sqlStatement <- "CREATE TABLE Analytics
                  (id INTEGER PRIMARY KEY, -- Autoincrement
                  date TEXT,
                  userGender VARCHAR(10),
                  users INT,
                  newUsers INT,
                  sessions INT,
                  bounces INT,
                  sessionDuration INT,
                  pageviews INT,
                  avgTimeOnPage DOUBLE,
                  bouncesHigh VARCHAR(15))"

dbExecute(myDB, sqlStatement)
```

```
## [1] 0
```

```

# Checking the result
dbListTables(myDB)
```

```
## [1] "Analytics"
```

```
dbListFields(myDB, "Analytics")
```

```
##  [1] "id"              "date"            "userGender"
##  [4] "users"           "newUsers"        "sessions"
##  [7] "bounces"         "sessionDuration" "pageviews"
## [10] "avgTimeOnPage"   "bouncesHigh"
```

```
dbReadTable(myDB, "Analytics")
```

```
##  [1] id              date            userGender      users
##  [5] newUsers        sessions        bounces         sessionDuration
##  [9] pageviews       avgTimeOnPage   bouncesHigh
## <0 rows> (or 0-length row.names)
```

Writing the data to the table using `dbWriteTable` and checking the result:

```
# Write to database
dbWriteTable(myDB,
             "Analytics",
             analytics,
             append = FALSE,
             overwrite = TRUE,
             row.names = FALSE)
# Check the result
dbReadTable(myDB,
            "Analytics") %>% head(n = 3)
```

```
##    date users newUsers sessions bounces sessionDuration pageviews
## 1 16211    12       11       13      10             425        18
## 2 16211    13       10       14      10             703        26
## 3 16216    12       10       15      10            2596        36
##   avgTimeOnPage userGender  bouncesHigh
## 1        85.200     female >=10 bounces
## 2        58.500       male >=10 bounces
## 3       123.619       male >=10 bounces
```

5.  Using a parameterized query to select those observations concerning male users:

```
# Reading lines of male users
criteria <- 'male'
sqlQuery <- "SELECT * FROM Analytics WHERE `userGender` = :x"
datSQL <- dbGetQuery(myDB,
                     sqlQuery,
                     params = list(x = criteria))
nrow(datSQL)
```

```
## [1] 75
```

```
datSQL %>% head(n = 3)
```

```
##    date users newUsers sessions bounces sessionDuration pageviews
## 1 16211    13       10       14      10             703        26
## 2 16216    12       10       15      10            2596        36
## 3 16217    14       11       21       8            5605        82
##   avgTimeOnPage userGender  bouncesHigh
## 1      58.50000       male >=10 bounces
## 2     123.61905       male >=10 bounces
## 3      91.90164       male  <10 bounces
```

Using a parameterized query and `dbBind` to select those observations that contain 13, 20 and 21 sessions:

```
# Reading lines with 13, 20, and 21 sessions

rs <- dbSendQuery(myDB,
                  "SELECT * FROM Analytics WHERE `sessions` = :x")
dbBind(rs, param = list(x = c(13, 20, 21)))
datSQL <- dbFetch(rs)
nrow(datSQL)
```

```
## [1] 16
```

```
datSQL %>% tail(n = 10)
```

```
##     date users newUsers sessions bounces sessionDuration pageviews
## 7  16311    11        7       13       9            7560        94
## 8  16324    12        9       13       9             200        25
## 9  16251    20       18       20      17             580        24
## 10 16274    20       18       20      15             443        34
## 11 16321    18       17       20      13            1891        44
## 12 16337    18       13       20      15            3395        41
## 13 16217    14       11       21       8            5605        82
## 14 16232    18       15       21      12            3005        39
## 15 16301    18       14       21      16             293        34
## 16 16325    20       18       21      17            6482        63
##    avgTimeOnPage userGender  bouncesHigh
## 7       93.38272       male  <10 bounces
## 8       16.58333     female  <10 bounces
## 9      144.75000     female >=10 bounces
## 10      31.57143       male >=10 bounces
## 11      78.87500       male >=10 bounces
## 12     161.66667       male >=10 bounces
## 13      91.90164       male  <10 bounces
## 14     166.88889       male >=10 bounces
## 15      22.61538       male >=10 bounces
## 16     154.28571       male >=10 bounces
```

```
dbClearResult(rs)
```

6. Tidy up:

Removing the *Analytics* table:

```
dbRemoveTable(myDB, "Analytics")
dbExistsTable(myDB, "Analytics")
```

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

Closing the connection:

```
# When disconnecting database vanishes
dbDisconnect(myDB)
```

### Checking the data

7. Since we are to check several conditions, we can define a function (lets call it `checkMe`) which holds all requirements. Then, we apply it to the dataset.

```
checkMe <- . %>%
  verify(nrow(.) > 10) %>%
  verify(mpg > 0) %>%
  insist(within_n_sds(2), mpg) %>%
  assert(in_set(0, 1), am, vs)

mtcars %>%
  checkMe
```

```
## Column 'mpg' violates assertion 'within_n_sds(2)' 2 times
##     verb redux_fn       predicate column index value
## 1 insist       NA within_n_sds(2)    mpg    18  32.4
## 2 insist       NA within_n_sds(2)    mpg    20  33.9
```

```
## Error: assertr stopped execution
```

8. In order to manipulate a variable using *dplyr* you can use the `mutate()` function. Just look at the following example. Note how we cannot see what the result of the other tests would have been if the first test had not failed. This is something, the *ensurer* package is capable of.

```
mtcarsManip <- mtcars[1:9, ] %>%
  mutate(mpg = replace(mpg, mpg == 14.3, -5)) %>%
  mutate(mpg = replace(mpg, mpg == 18.1, -150)) %>%
  mutate(vs = replace(vs, mpg == -5, 2))

mtcarsManip %>%
  checkMe
```

```
## verification [nrow(.) > 10] failed! (1 failure)
##
##     verb redux_fn    predicate column index value
## 1 verify       NA nrow(.) > 10     NA     1    NA
```

```
## Error: assertr stopped execution
```

9. In the following we test two conditions simultaneously. First, we use `ensures_that()` in order to prespecify the expectations. Then we combine them using `ensure()`.

```
moreThan10Rows <- ensures_that(NROW(.) > 10)
mpgAllPos <- ensures_that(all(.[, names(.) == "mpg"] > 0))
# Using the original dataset
mtcars %>% ensure(+moreThan10Rows, +mpgAllPos) %>% head(n = 5)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

```
# Using the manipulated dataset
mtcarsManip %>% ensure(+moreThan10Rows, +mpgAllPos) %>% head(n = 5)
## Error: conditions failed for call 'mtcarsManip %>% ensure(+moreTh .. s,
    +mpgAllPos) %>% head(n = 5)':
##   * NROW(.) > 10
##   * all(.[, names(.) == "mpg"] > 0)
```

10. Again we use `ensures_that()` in order to specify the additional requirements. Then we add them to our call to `ensure()`.

```
smaller2Sd <- ensures_that(all(.[, names(.) == "mpg"] <
                                 (mean(.[, names(.) == "mpg"]) +
                                    2 * sd(.[, names(.) == "mpg"]))))
larger2Sd <- ensures_that(all(.[, names(.) == "mpg"] >
                                (mean(.[, names(.) == "mpg"]) -
                                   2 * sd(.[, names(.) == "mpg"]))))

mtcars %>% ensure(+moreThan10Rows, +mpgAllPos, +smaller2Sd, +larger2Sd)
## Error: conditions failed for call 'mtcars %>% ensure(+moreThan10Rows,
##        +mpgAllPos, +smaller2Sd, +larger2Sd)':
##          * all(.[, names(.) == "mpg"] < (mean(.[, names(.) == "mpg"]) +
```

# 4 Tidy Data

In the information age, data quality is a main determinant for the economic success of a company. Incorrect, inaccurate or missing data lead to incomplete or faulty information and thus to inefficient or wrong decisions. The quality of statistical data analysis is directly connected to the data quality (garbage in - garbage out principle). In the following sections, which are partially influenced by Hadley Wickham's concept of *Tidy Data*[16], we will give you some advice on how to collect, clean and handle your data appropriately.

## 4.1 Things to Consider when Collecting Data Yourself

As mentioned before, data is the basis of statistical analysis. One can distinguish between data obtained by *primary data* collection (data is collected particularly for one analysis, e.g., through a self-administered survey) and *secondary data* collection (existing data from, e.g., databases like SourceOECD is used). Data from primary data collection is tailored specifically to the requirements of the corresponding analysis and is always up to date. Drawbacks are the (possibly) very high costs and resources needed for planning and executing the collection. Depending on the collection method, the time period until the data is accessible can vary considerably, including up to several weeks. Hence, one should rely on existing data if it meets the requirements and is well-suited to answer the research question. Complementary to the section about representativity (section **??**), the following section states some ideas about sampling.

Before the actual survey is conducted, lots of theoretical work has to be done. Questions that need to be answered are (among others) the following:

- Who do you want to make inferences about? Which population are you aiming at?
- In your sample, how many people do you want to ask? (This should be answered by a power analysis.)
- How can you reach them?
- And who are you going to be able to state things about? Who is going to answer?

Concepts that play a role in the context of data quality are for example selection bias, missing values and weighting factors. Selection bias can occur when people in an experiment decide themselves if they belong to the treatment or control group. Or if people assigned to one group drop out more often than people from another group. This draws our attention to the issue of missing values. If there are observations missing systematically with respect to a characteristic interesting for analysis, results can be considerably biased, even if the amount of missings is really small. The following example illustrates a mechanism for such a biasing of results:

During World War II, the British airforce wanted to increase safety for its pilots by reinforcing the planes' outer skin. Hence, they searched for typical damage of the planes that had returned. Surprisingly, it seemed that the Germans missed precisely that part of the aircraft where the tank was. How come? Well, the British almost made a crucial mistake. They drew conclusions based on a biased sample because they only examined the returned planes but not those which didn't return. The planes that did not return were those hit at the most sensitive part of a plane, the tank. So, from a survey point of view, they basically made two mistakes: First, they wanted to infere

[16]Wickham, Hadley: Tidy Data. Journal of Statistical Software 59 (2014), available at http://www.jstatsoft.org/article/view/v059i10

about a population (all planes) whose members did not all have the chance to enter in the sample (the crashed planes were excluded from the beginning). And secondly, the planes were excluded in a systematic way. They were differing systematically from those which returned with respect to the characteristic crucial for analysis, the position of bullet holes.

There are two things that can be done when bias is to be expected. The first and probably best thing to do is to avoid bias in the first place. This can partially be achieved by thoroughly answering the questions presented above beforehand and by planning the study accordingly. The second thing one can do is to use weigths in order to unbias results. Hence, the British airforce could have tried to examine as many crashed planes as possible and given them a higher weight.

Concerning the planning of an experimental design, similar care must be taken. In order to understand the basics, lets consider an experiment in a (from a statistical point of view) perfect world. If for example the effect of a certain treatment is to be examined we

- would randomly assign people to the treatment and the control group,
- observe them some time before the treatment,
- then set the treatment,
- and observe both groups for some while after the treatment was set.

Why is this important? Firstly, by examining both groups before the treatment, we can check whether or not they differ considerably. If they do, differences between the groups cannot be assigned to the treatment alone later on and new groups have to be found. Additionally, we want to observe both groups before and after the treatment in order to investigate whether or not there are so-called time effects, i.e., whether or not some changes are simply caused by the time passing. Those and other things need to be considered when planning an experiment. And, of course, in our perfect world, all persons observed will happily take part in our experiment and stick with us until we don't need them any more. In reality this unfortunately is rarely the case. Hence, there are some rules for survey design that help to convince people to answer the questions they are asked.

## 4.2 Tidy Data

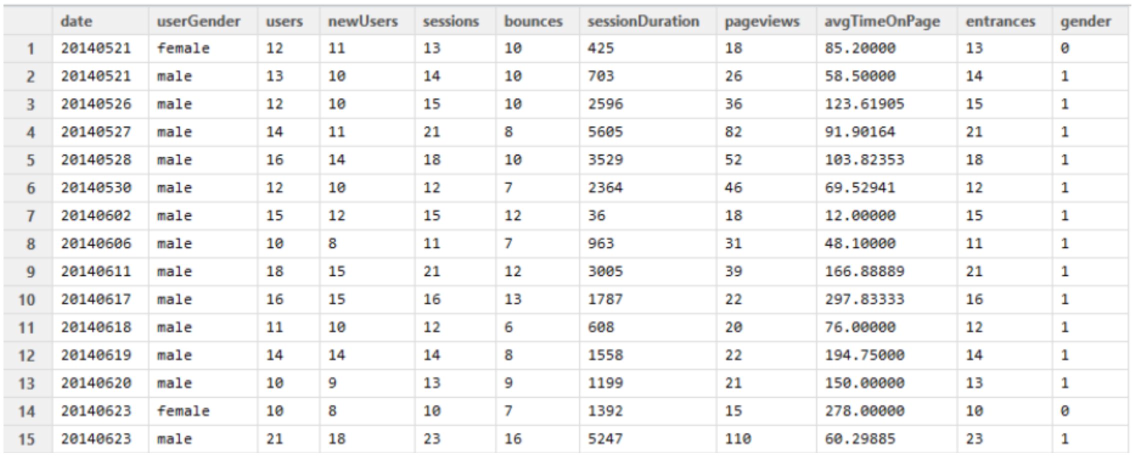
So far we have looked at data that has been collected in a methodologically clean manner. We now turn to concrete rules for the form in which data is stored. Since statistical analysis nowadays is conducted exclusively via computer, data have to be digitalized first (if they haven't been collected electronically). This usually involves miniaturisation of the data, which is a form of coding that facilitates analysis and saves storage space. Through coding, single variables are defined that contain – if possible – exclusively numeric values. In order to obtain tidy data, it makes absolutely sense to produce a coding scheme in advance (this requires the conceptualization of the entire analysis at an early stage).

According to Wickham, tidying means giving datasets a structure in order to facilitate subsequent analysis. In *tidy data*:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Fixed variables, that describe for example the experimental set-up, come first in a dataset. Variables that are measured follow. *Messy data* is anything else.

Figure 6 shows as example of a tidy dataset with 15 observations of a Google Analytics dataset. The observations are identified by the fixed columns `date` and `gender` which come first. Then the different measured variables follow. Each row holds an observation and each column one variable. It contains tracking data about the behavior of visitors of the INWT Statistics website. Observations are grouped by date and gender, hence, each row contains a unique date-gender-combination. So the first line of the data can be read as follows: On May $21^{st}$ in 2014, 12 female users visited the INWT website. Among them were 11 new users, i.e., their cookie has not been recognized. All in all they started 13 sessions of which there were 10 bounces. This means that they stayed only on one page, leaving it unclear how long they stayed or where they went afterwards. The 13 sessions lasted a total of 425 seconds. There have been 18 pageviews with an average time on page of 85.2 seconds. The variable `entrances` contains the same information as the variable `sessions`. Same holds for the variable `gender` which is, except for the coding, the same as `userGender`.



| | date | userGender | users | newUsers | sessions | bounces | sessionDuration | pageviews | avgTimeOnPage | entrances | gender |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 20140521 | female | 12 | 11 | 13 | 10 | 425 | 18 | 85.20000 | 13 | 0 |
| 2 | 20140521 | male | 13 | 10 | 14 | 10 | 703 | 26 | 58.50000 | 14 | 1 |
| 3 | 20140526 | male | 12 | 10 | 15 | 10 | 2596 | 36 | 123.61905 | 15 | 1 |
| 4 | 20140527 | male | 14 | 11 | 21 | 8 | 5605 | 82 | 91.90164 | 21 | 1 |
| 5 | 20140528 | male | 16 | 14 | 18 | 10 | 3529 | 52 | 103.82353 | 18 | 1 |
| 6 | 20140530 | male | 12 | 10 | 12 | 7 | 2364 | 46 | 69.52941 | 12 | 1 |
| 7 | 20140602 | male | 15 | 12 | 15 | 12 | 36 | 18 | 12.00000 | 15 | 1 |
| 8 | 20140606 | male | 10 | 8 | 11 | 7 | 963 | 31 | 48.10000 | 11 | 1 |
| 9 | 20140611 | male | 18 | 15 | 21 | 12 | 3005 | 39 | 166.88889 | 21 | 1 |
| 10 | 20140617 | male | 16 | 15 | 16 | 13 | 1787 | 22 | 297.83333 | 16 | 1 |
| 11 | 20140618 | male | 11 | 10 | 12 | 6 | 608 | 20 | 76.00000 | 12 | 1 |
| 12 | 20140619 | male | 14 | 14 | 14 | 8 | 1558 | 22 | 194.75000 | 14 | 1 |
| 13 | 20140620 | male | 10 | 9 | 13 | 9 | 1199 | 21 | 150.00000 | 13 | 1 |
| 14 | 20140623 | female | 10 | 8 | 10 | 7 | 1392 | 15 | 278.00000 | 10 | 0 |
| 15 | 20140623 | male | 21 | 18 | 23 | 16 | 5247 | 110 | 60.29885 | 23 | 1 |

**Figure 5** – A subset from the Google Analytics dataset

A description of the data could take the following form and is fairly helpful for analysis:

| **Variable** | **Description** | **Measurement scale** | **Values** |
|---|---|---|---|
| `date` | date of access | metric | |
| `userGender` | gender of user | nominal | female, male |
| `users` | number of users | metric | |
| `newUsers` | number of new users | metric | |
| `sessions` | number of sessions | metric | |
| `bounces` | number of bounces | metric | |
| `sessionDuration` | total duration of user sessions (seconds) | metric | |
| `pageviews` | total number of pageviews | metric | |
| `avTimeOnPage` | average time on page (seconds) | metric | |
| `gender` | gender of user | nominal | 0 (female), 1 (male) |

Note the definition of the variable `gender`, which holds the values 0 (for female) and 1 (for male).

### 4.3 The tidyr package

In order to obtain tidy data, Wickham's package `tidyr` provides some basic functions. It is the evolution of `reshape2` and designed for data tidying (not general reshaping or aggregating). The facts and examples below are taken from the package's vignette which additionally contains code examples and links to github repositories with real world tidying projects. Wickham basically describes five problems that render data messy:

- Column headers are values, not variable names.
- Multiple variables are stored in one column.
- Variables are stored in both rows and columns.
- Multiple types of observational units are stored in the same table.
- A single observational unit is stored in multiple tables.

There are three basic functions that help tidying data: `gather()`, `spread()`, and `separate()`.

#### 4.3.1 gather()

With `gather()` you can transform data from wide into long format. It helps to put non-variable (fixed) columns into key-value combinations. In this case the values from the columns `treatmenta` and `treatmentb` are combined into one column `n` and their labels are collected in the additional column `treatment`.

```r
# Messy Data
library(readr)
preg <- read_csv(file = paste0(pathData, "preg.csv"))
preg
```

```
## # A tibble: 3 x 3
##   name         treatmenta treatmentb
##   <chr>             <int>      <int>
## 1 John Smith           NA         18
## 2 Jane Doe              4          1
## 3 Mary Johnson          6          7
```

```r
# Gathered (tidy) Data
library(tidyr)
pregTidy <- preg %>%
  gather(treatment, n, treatmenta:treatmentb)
pregTidy
```

```
## # A tibble: 6 x 3
##   name         treatment      n
##   <chr>        <chr>      <int>
## 1 John Smith   treatmenta    NA
## 2 Jane Doe     treatmenta     4
## 3 Mary Johnson treatmenta     6
## 4 John Smith   treatmentb    18
## 5 Jane Doe     treatmentb     1
## 6 Mary Johnson treatmentb     7
```

### 4.3.2 spread()

spread() is the inverse function of gather. You can use it to transform data from long into wide format. It spreads the identifier of observational type into columns, i.e., if for example a column contains variables' names, those names are used as new column names and the observational types are allocated accordingly. In the following dataset the column element contains the identifiers tmax and tmin. The spread() function spreads the values they identify into two accordingly named columns.

```
# Messy Data
weather <- read_csv(file = paste0(pathData, "weather2.csv"))
head(weather)
```

```
## # A tibble: 6 x 6
##   id        year month element day   value
##   <chr>    <int> <int> <chr>   <chr> <dbl>
## 1 MX17004   2010    12 tmax    d1     29.9
## 2 MX17004   2010    12 tmin    d1     13.8
## 3 MX17004   2010     2 tmax    d2     27.3
## 4 MX17004   2010     2 tmin    d2     14.4
## 5 MX17004   2010    11 tmax    d2     31.3
## 6 MX17004   2010    11 tmin    d2     16.3
```

```
# Spread (tidy) Data
weatherTidy <- weather %>% spread(element, value)
head(weatherTidy)
```

```
## # A tibble: 6 x 6
##   id        year month day    tmax  tmin
##   <chr>    <int> <int> <chr> <dbl> <dbl>
## 1 MX17004   2010     1 d30    27.8  14.5
## 2 MX17004   2010     2 d11    29.7  13.4
## 3 MX17004   2010     2 d2     27.3  14.4
## 4 MX17004   2010     2 d23    29.9  10.7
## 5 MX17004   2010     2 d3     24.1  14.4
## 6 MX17004   2010     3 d10    34.5  16.8
```

### 4.3.3 separate()

separate() helps to split compound columns, i.e., if two kind of observational types are contained in one column, e.g., in form of alphanumeric codes. separate() extracts both pieces of information and stores them in separate columns. In the following example, the observational types gender and age are stored in one column demo.

```
tb <- read_csv(file = paste0(pathData, "tb2.csv"))
head(tb)
```

```
## # A tibble: 6 x 4
##   iso2   year demo      n
##   <chr> <int> <chr> <int>
## 1 AD     2005 m04      0
## 2 AD     2006 m04      0
## 3 AD     2008 m04      0
## 4 AE     2006 m04      0
## 5 AE     2007 m04      0
## 6 AE     2008 m04      0
```

`separate()` takes its values, splits them and stores them in the new variables `gender` and `age`. In addition we have to tell the position where `demo` should be split: After the 1st character.

```
# Separated (tidy) Data
tbTidy <- tb %>%
  separate(demo, c("gender", "age"), 1)
head(tbTidy)
```

```
## # A tibble: 6 x 5
##   iso2   year gender age       n
##   <chr> <int> <chr>  <chr> <int>
## 1 AD     2005 m      04        0
## 2 AD     2006 m      04        0
## 3 AD     2008 m      04        0
## 4 AE     2006 m      04        0
## 5 AE     2007 m      04        0
## 6 AE     2008 m      04        0
```

Furthermore, `dplyr` is used in order to simplify, sort, and rearrange the data. Often it is necessary to rename variables, generate new variables, etc.

### 4.3.4 Split a Dataset into Multiple Datasets

Sometimes a data table needs to be broken up into several tables if the dataset contains values collected at multiple levels. The following dataset contains on the one hand information about the `rank` of certain songs in the charts in an indicated `week`. On the other hand it contains information about the song's name (`track`), the `artist`, the `time` (duration) and the release `date`. Note that the rank dataset and the song dataset are measured on different levels. Consequently, equipped with a song id, the data can be stored in two different tables: one containing the song information and one containing information about the rankings.

```
# Combined Messy Data
billboard2 <- read_csv(file = paste0(pathData, "billboard2.csv"))
head(billboard2)
```

```
## # A tibble: 6 x 6
##   artist       track                  time    week  rank date
##   <chr>        <chr>                  <time> <int> <int> <date>
## 1 2 Pac        Baby Don't Cry (Keep... 04:22      1    87 2000-02-26
## 2 2Ge+her      The Hardest Part Of ... 03:15      1    91 2000-09-02
## 3 3 Doors Down Kryptonite             03:53      1    81 2000-04-08
## 4 3 Doors Down Loser                    04:24      1     76 2000-10-21
## 5 504 Boyz      Wobble Wobble            03:35      1     57 2000-04-15
## 6 98^0          Give Me Just One Nig... 03:24      1     51 2000-08-19
```

In the following code snippets you don't need to understand each and every line since some of the functions are only introduced in section 5. The important thing is that you understand why we split the data into several tables.

```
songInfo <- billboard2 %>%
  select(artist, track, time, date) %>%
  unique() %>%
  arrange(track) %>%
  mutate(song_id = row_number())

head(songInfo)
```

```
## # A tibble: 6 x 5
##   artist       track                    time   date       song_id
##   <chr>        <chr>                    <time> <date>       <int>
## 1 Nelly        (Hot S**t) Country G... 04:17  2000-04-29       1
## 2 Nu Flavor    3 Little Words           03:54  2000-06-03       2
## 3 Jean, Wyclef 911                      04:00  2000-10-07       3
## 4 Brock, Chad  A Country Boy Can Su... 03:54  2000-01-01       4
## 5 Clark, Terri A Little Gasoline        03:07  2000-12-16       5
## 6 Son By Four  A Puro Dolor (Purest... 03:30  2000-04-08       6
```

```
rankInfo <- billboard2 %>%
  # Add song id:
  left_join(songInfo, by = c("artist", "track", "time", "date")) %>%
  select(song_id, week, rank) %>%
  arrange(song_id, week)

head(rankInfo)
```

```
## # A tibble: 6 x 3
##   song_id  week  rank
##     <int> <int> <int>
## 1       1     1   100
## 2       1     2    99
## 3       1     3    96
## 4       1     4    76
## 5       1     5    55
## 6       1     6    37
```

For analysis it could, however, be necessary to merge datasets back together. Also note that it is sometimes easier to input *messy* data. This is absolutely okay if tidying takes place afterwards.

## 4.4 Differences in Scales of Measurement

After coding, the data consists mainly of numeric values. Hence, it is theoretically possible to conduct numerous arithmetic operations (e.g. addition, multiplication, . . . ). However, depending on the variable's scale of measurement, not all of the operations make sense. Determination of the scale of measurement is thus of utmost importance:

| Measurement scale | Allowed operations | Examples |
|---|---|---|
| nominal | $=, \neq$ | gender, state |
| ordinal | $=, \neq, <, >$ | satisfaction on a 5-point scale, (Highschool-)grades |
| metric | $=, \neq, <, >, +, -, *, :$ | income, response time |

Like other statistical packages, the common functions in R process the variables according to their scale of measurement. If one tries to apply a function requiring a certain scale of measurement to a variable of another scale, R issues at least a warning[17]. But be careful: Some information consists of numbers but is neither numeric nor ordinal, e.g., german zip codes. So they must not be treated as metric – or do you know how to interpret the "mean zip code" of a sample?

### Likert Scales

A common example for a Likert scale is a 5-point scale measuring satisfaction. A typical response scheme is "I totally agree" to "I do not agree at all". Even though the according characteristics are methodologically classified as ordinal, in practice they are often treated as metric.

Additionally, it makes sense to distinguish between *discrete* and *continuous* characteristics. Discrete characteristics can take on a finite number or at least a countable number of values (typically characteristics measured on nominal or ordinal scale are considered discrete). Continuous variables can take on an infinite number of values within one interval.

Practically the distinction is not that easy. Body height for example could theoretically be measured on a continuous scale. In practice, however, it is mostly measured only up to a precision of a centimeter.

[17]This may not apply to all of the functions available.

### 4.5 Missing Values and Special Values

Concerning the validity of data, the following situations are distinguished:

a) valid value: characteristic is available and has a value in the expected range
b) not specified: characteristic is defined for statistical unit, but not observed (missing value)
c) not applicable: characteristic is not defined for this statistical unit (e.g., after filter question)

The coding scheme should contain guidance for the treatment of cases b) and c). Missing values (case b) can for example be coded with "-1" and values not applicable (case c) with "-2". One should pay attention not to mark cases b) and c) with values that could apply to the valid values of the respective variable (for example a common number to indicate missing values is "99", which can be confounded with valid observations of numerous variables). If possible, cases b) and c) should be coded differently.

Furthermore, `R` provides some special values for this purpose. If for example no distinction needs to be made between values not specified (case b) and values not applicable (case c), a missing observation for a variable can simply be coded with `"NA"`. Almost all functions contain special routines about how to handle those missing values. In addition there is `"NaN"` ("Not a Number") and `"Inf"` (infinity):

```r
0 / 0
```

```
## [1] NaN
```

```r
log(0)
```

```
## [1] -Inf
```

## 4.6 R: Importing and Viewing Data

Following all this theory about data management we want to start working with data ourselves. Since we cannot collect data here in this course, we need to rely on data already collected. So please save the Google Analytics raw data in the folder you saved as `pathData`.

To read in the *.csv* file, we can use `read_csv()` from the package `readr`. This package should be preferred over the `base` package as already mentioned.

```
read_csv(file = paste0(pathData, "AnalyticsRaw.csv"))
```

To ease up dealing with the Google Analytics dataset, we assign the name `analyticsRaw` to it. It is by the way possible to repeat the previously entered command by using the arrowkey ↑.

```
analyticsRaw <- read_csv(file = paste0(pathData, "AnalyticsRaw.csv"))
```

If you executed the code above, up in the right corner of RStudio, in the *Environment* tab, the `analyticsRaw` table should occur in your workspace.

### 4.6.1 Subsetting

In section "R 101" we learned several ways to access data stored in a dataset: If the name of the dataset is succeeded by a $ sign and the name of a variable contained in the dataset, R returns the content of that variable. As we have seen before, the function `dim()` returns the dimension of a dataset. It helps to subset parts of the data, i.e., to access single elements contained in the dataset. To this end, the name of the dataset is followed by `[x, y]`, where `x` denotes the number of the corresponding row (starting at 1) and `y` denotes the number of the corresponding column (starting at 1 as well). If you omit one of the indices, the whole row resp. column is returned. Instead of a single number for a row or column, a whole range of rows or columns can be accessed with `a:b` (remind yourself of the selection of the $2^{nd}$ to $4^{th}$ element of the `rank` vector above)[18]. Additionally, there is the `subset()` function from the `base` package and the `filter()` function from the `dplyr` package. But more on that later.

*Reminder:* R is case-sensitive. So watch out for upper and lower case.

Look at the variable containing the number of bounces via the $ sign or the subsetting operator.

```
analyticsRaw$bounces
```

```
##   [1] 10 10 10  8 10  7 12  7 12 13  6  8  9  7 16  7 13 11  7  7 17 12  9
##  [24] 13 14 15 12 10  9  9 11 10  6  7  7 11 11 11  8  8 11  9 10 15 13  7
##  [47]  8  9  8  6  9 14  7  6 14  9 16 11 11 11 10 10 11 11 13  5 10  9  8
##  [70] 10  8 12 14 13 11  6 13  9 11 17  7  8 11  9 15  6 15 14  8 19 13 19
##  [93] 11 15  5 12 11  4  4  8  9 12
```

[18]Some people like to use the command `attach()` in order to integrate the dataset directly into the search path. In consequence all variables can be selected immediately by their names. Even though the idea of getting rid of that cumbersome $ sign may be tempting, you must be warned that using `attach()` can lead to quite some confusion. This is especially the case if there are variables from different datasets but with the same name. The `search()` function lists the objects contained in the search path. In order to remove a dataset from the search path, the `detach()` command can be used.

```r
analyticsRaw[, 5]
```

```
## # A tibble: 102 x 1
##    bounces
##      <int>
##  1      10
##  2      10
##  3      10
##  4       8
##  5      10
##  6       7
##  7      12
##  8       7
##  9      12
## 10      13
## # ... with 92 more rows
```

Similarly, columns can be selected by giving the variable's name:

```r
analyticsRaw[, "bounces"]
```

```
## # A tibble: 102 x 1
##    bounces
##      <int>
##  1      10
##  2      10
##  3      10
##  4       8
##  5      10
##  6       7
##  7      12
##  8       7
##  9      12
## 10      13
## # ... with 92 more rows
```

And selecting multiple columns:

```r
analyticsRaw[, c(1, 5)]
```

```
## # A tibble: 102 x 2
##        date bounces
##       <int>   <int>
##  1 20140521      10
##  2 20140521      10
##  3 20140526      10
##  4 20140527       8
##  5 20140528      10
##  6 20140530       7
##   7 20140602       12
##   8 20140606        7
##   9 20140611       12
##  10 20140617       13
## # ... with 92 more rows
```

```
analyticsRaw[, c("date", "bounces")]
```

```
## # A tibble: 102 x 2
##         date bounces
##        <int>   <int>
##   1 20140521      10
##   2 20140521      10
##   3 20140526      10
##   4 20140527       8
##   5 20140528      10
##   6 20140530       7
##   7 20140602      12
##   8 20140606       7
##   9 20140611      12
##  10 20140617      13
## # ... with 92 more rows
```

Using the `dim()` function to check the dimensions:

```
dim(analyticsRaw)
```

```
## [1] 102  10
```

The same information can be extracted using the `nrow()` function (giving the number of rows) and the `ncol()` function (giving the number of columns).

Choosing the element in the $33^{rd}$ row and the $4^{th}$ column:

```
analyticsRaw[33, 4]
```

```
## # A tibble: 1 x 1
##   sessions
##      <int>
## 1       11
```

Choosing the first ten elements in the $6^{th}$ column.

```
analyticsRaw[1:10, 6]
```

```
## # A tibble: 10 x 1
##    sessionDuration
##              <int>
##   1            425
##   2            703
##  3             2596
##  4             5605
##  5             3529
##  6             2364
##  7               36
##  8              963
##  9             3005
## 10             1787
```

A more complex form of subsetting is the selection of certain variables for observations that satisfy certain conditions. We want, for example, to select the observations on the variables `newUsers` and `sessionDuration` where the number of new users lies within the (closed) interval ranging from 10 to 12 and the session duration does not exceed 1778 seconds. This can be achieved either by using the `subset()` function or the subsetting operator `[ ]`. Several conditions can be linked either by a `&` sign (logical "and"), meaning that both conditions must hold, or a `|` sign (logical "or"), denoting that at least one of the conditions must hold. More subsetting can be achieved using `dplyr`.

```
subset(analyticsRaw,
       newUsers >= 10 & newUsers <= 12 & sessionDuration <= 1778,
       select = c(newUsers, sessionDuration))

analyticsRaw[analyticsRaw$newUsers >= 10 &
               analyticsRaw$newUsers <= 12 &
               analyticsRaw$sessionDuration <= 1778,
             c("newUsers", "sessionDuration")]
```

### 4.6.2 Getting an Overview of Your Dataset

Principally you are well-advised to get an overview of your data and its structure before doing anything else. Mere collection of data is not all there is. In order to do a sound analysis, you have to know your data by heart. Already during the collection process it makes sense to permanently check your data. Rather often useless information is collected whereas essential information is forgotten to be tracked. If you don't see your data answering the questions you want to answer, nobody else will. Anyways, more on that later, let's look at the Analytics data now.

Important commands are `summary()`, `str()`, and `names()` – but there are many more.

```
summary(analyticsRaw)
```

```
##       date               users           newUsers         sessions    
##  Min.   :20140521   Min.   :10.00   Min.   : 7.00   Min.   :10.00  
##  1st Qu.:20140704   1st Qu.:11.00   1st Qu.:10.00   1st Qu.:12.00  
##  Median :20140806   Median :13.00   Median :11.00   Median :15.00  
##  Mean   :20140777   Mean   :14.09   Mean   :11.73   Mean   :15.48  
##  3rd Qu.:20140910   3rd Qu.:16.00   3rd Qu.:13.00   3rd Qu.:18.00  
##  Max.   :20140930   Max.   :26.00   Max.   :23.00   Max.   :33.00  
##     bounces       sessionDuration   entrances          gender      
##  Min.   : 4.00   Min.   :   36   Min.   :10.00   Min.   :0.0000  
##  1st Qu.: 8.00   1st Qu.:  694   1st Qu.:12.00   1st Qu.:0.0000
##  Median :10.00   Median : 1778   Median :15.00   Median :1.0000
##  Mean   :10.29   Mean   : 2541   Mean   :15.48   Mean   :0.7353
##  3rd Qu.:12.00   3rd Qu.: 3365   3rd Qu.:18.00   3rd Qu.:1.0000
##  Max.   :19.00   Max.   :18964   Max.   :33.00   Max.   :1.0000
##    pageviews       avgTimeOnPage
##  Min.   : 13.00   Min.   : 12.00
##  1st Qu.: 21.25   1st Qu.: 60.14
##  Median : 30.00   Median :102.36
##  Mean   : 39.63   Mean   :120.13
##  3rd Qu.: 43.50   3rd Qu.:165.58
##  Max.   :207.00   Max.   :384.40
```

The output of `summary()` depends on the object's class. The summary of numeric variables (e.g. `sessions`) gives the minimum, 1st quartile, median, mean, 3rd quartile and maximum. For factors we get counts of their levels.

```
str(analyticsRaw)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    102 obs. of  10 variables:
##  $ date           : int  20140521 20140521 20140526 20140527 20140528
    20140530 20140602 20140606 20140611 20140617 ...
##  $ users          : int  12 13 12 14 16 12 15 10 18 16 ...
##  $ newUsers       : int  11 10 10 11 14 10 12 8 15 15 ...
##  $ sessions       : int  13 14 15 21 18 12 15 11 21 16 ...
##  $ bounces        : int  10 10 10 8 10 7 12 7 12 13 ...
##  $ sessionDuration: int  425 703 2596 5605 3529 2364 36 963 3005 1787 ...
##  $ entrances      : int  13 14 15 21 18 12 15 11 21 16 ...
##  $ gender         : int  0 1 1 1 1 1 1 1 1 1 ...
##  $ pageviews      : int  18 26 36 82 52 46 18 31 39 22 ...
##  $ avgTimeOnPage  : num  85.2 58.5 123.6 91.9 103.8 ...
##  - attr(*, "spec")=List of 2
##   ..$ cols   :List of 10
##   .. ..$ date           : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ users          : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ newUsers       : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ sessions       : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ bounces        : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ sessionDuration: list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ entrances      : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##   .. ..$ gender         : list()
##   .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##    .. ..$ pageviews      : list()
##    .. .. ..- attr(*, "class")= chr  "collector_integer" "collector"
##    .. ..$ avgTimeOnPage  : list()
##    .. .. ..- attr(*, "class")= chr  "collector_double" "collector"
##    ..$ default: list()
##    .. ..- attr(*, "class")= chr  "collector_guess" "collector"
##    ..- attr(*, "class")= chr "col_spec"
```

The `str()` command (str for structure) prints quite a good overview of the dataset. The first line of structure includes the object's class, number of observations and variables. Below, the included variables are listed as well as their class and a preview of their content.

A somewhat simpler overview over the variables contained in the data can be obtained by the `names()` function.

```
names(analyticsRaw)
```

```
##  [1] "date"            "users"           "newUsers"
##  [4] "sessions"        "bounces"         "sessionDuration"
##  [7] "entrances"       "gender"          "pageviews"
## [10] "avgTimeOnPage"
```

Sometimes you may feel like you have to look at your data as a whole. Either to get further ideas about your analysis or just because you have been working with other software where this was possible... Here again, `read_csv()` from the `readr` package again has an advantage over `read.csv()` from the `base` package since it attributes the classes "tbl_df", "tbl", and "data.frame" to the data. So if the package `dplyr` is loaded and the data is of class "tbl_df", the first ten lines are printed to the console by typing the dataset's name (this is similar to the `base` command `head()`). Note that the number of variables displayed varies according to the width of the console.

```
analyticsRaw
```

```
## # A tibble: 102 x 10
##       date users newUsers sessions bounces sessionDuration entrances gender
##      <int> <int>    <int>    <int>   <int>           <int>     <int>  <int>
##   1 2.01e7    12       11       13      10             425        13      0
##   2 2.01e7    13       10       14      10             703        14      1
##   3 2.01e7    12       10       15      10            2596        15      1
##   4 2.01e7    14       11       21       8            5605        21      1
##   5 2.01e7    16       14       18      10            3529        18      1
##   6 2.01e7    12       10       12       7            2364        12      1
##   7 2.01e7    15       12       15      12              36        15      1
##   8 2.01e7    10        8       11       7             963        11      1
##   9 2.01e7    18       15       21      12            3005        21      1
##  10 2.01e7    16       15       16      13            1787        16      1
## # ... with 92 more rows, and 2 more variables: pageviews <int>,
## #   avgTimeOnPage <dbl>
```

There are two things we notice: First, the variable `entrances` seems to be identical to `sessions` and can be dropped.

```
analyticsRaw <- analyticsRaw[, -which(names(analyticsRaw) == "entrances")]
```

Secondly, the `gender` variable is rather uninformative. This is due to the following: When data is read with `read_csv()` from the package `readr`, the data is imported as it is. This means that for example a variable for gender, which is coded 0 for female and 1 for male, is detected as being integer. Since the class integer implies that the values are on an interval scale, but gender however is nominal, the variable's class should be changed to `factor`. This is achieved using the function `factor()`. Now the predefined labels are shown in order to enhance readability of the output. Analogously, you can define variables to have ordered scale with the function `ordered()`.

See the following example as illustration:

```
class(analyticsRaw$gender)
```

```
## [1] "integer"
```

```
analyticsRaw$userGender <- factor(analyticsRaw$gender)
class(analyticsRaw$userGender)
```

```
## [1] "factor"
```

```
levels(analyticsRaw$userGender)
```

```
## [1] "0" "1"
```

```
levels(analyticsRaw$userGender) <- c("female", "male")
head(analyticsRaw$userGender)
```

```
## [1] female male   male   male   male   male
## Levels: female male
```

Now we can kick out the redundant `gender` variable and save the dataset as .csv file called "Analytics".

```
analyticsRaw <- analyticsRaw[, -which(names(analyticsRaw) == "gender")]
write.csv(analyticsRaw,
          file = paste0(pathData, "Analytics.csv"),
          row.names = FALSE)
```

Note that, when using `read.csv()` or `read.csv2()` from the `base` package, `R` tries to guess each column's corresponding scale of measurement. Then, alpha-numeric columns are converted to factors by default. This conversion, though mostly helpful, can be prevented by setting the parameter `as.is = TRUE`. Generally it makes sense to check the scale of measurement after the data has been read in. In multiple cases variables coded as numeric or integer have to be subsequently defined as factors.

**Further Reading**

- Wickham, H., & Grolemund, G. (2017). R for Data Science. Available under r4ds.had.co.nz/
- Wickham, H. (2014): Tidy data. The Journal of Statistical Software, vol. 59, 2014. Available under vita.had.co.nz/papers/tidy-data.html
- Wickham, H.: Tidy Data. `tidyr` vignette, available under cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html

**Exercises**

In the table below, we list some commands that may be useful for exploring your workspace and data structure. We did not fill in the "Output" column, though. So go ahead and fill in the blanks.

| Command | Output |
|---|---|
| `ls()` | |
| `class()` | |
| `dim()` | |
| `object.size()` | |
| `summary()` | |
| `str()` | |
| `names()` | |
| `head()` | |
| `tail()` | |
| `rm()` | |

Keep working with the `analyticsRaw` dataset. By using different ways of how to select a certain element from a data frame (remember $ sign, `dim()`, `[x, y]...`):

1. Select the element in the $3^{rd}$ row and $7^{th}$ column,
2. select the column containing the variable `pageviews`,
3. select the element in the last row and the last column in the dataset (imagine that the number of rows and number of columns is not know or could change from time to time, but your code should work anyway),
4. select all rows containing information on male users,
5. and select the variables `userGender` and `date` where the number of sessions is between 12 and 18.

**Solutions**

1.

```
analyticsRaw[3, 7]
```

```
## # A tibble: 1 x 1
##   gender
##    <int>
## 1      1
```

2.

```
analyticsRaw[["pageviews"]]
```

```
##   [1]  18  26  36  82  52  46  18  31  39  22  20  22  21  15 110  22  40
##  [18]  41  24  25  24  20  13  30  18  45  37  35  36  13  33  25  18  30
##  [35]  21  18  33  77  31  29  25  83  26  34  26  15  27  21  15  21  32
##  [52]  40  31  24 125 115  34  26 207 106  80  34  76  22  27  49  48  94
##  [69]  30  21  33  23  29  44  42  46  73  25  88  63  23  21  34  30  75
##  [86]  21  68  27  19  83  52  61  34  41  19  37  20  20  20  17  15  29
```

```
##
## # Or:
## analyticsRaw$pageviews
```

3.

```
analyticsRaw[nrow(analyticsRaw), ncol(analyticsRaw)]
```

```
## # A tibble: 1 x 1
##   userGender
##   <fct>
## 1 male
```

4.

```
analyticsRaw[analyticsRaw$userGender == "male", ]
```

```
## # A tibble: 75 x 10
##      date users newUsers sessions bounces sessionDuration gender pageviews
##     <int> <int>    <int>    <int>   <int>           <int>  <int>     <int>
##  1 2.01e7    13       10       14      10             703      1        26
##  2 2.01e7    12       10       15      10            2596      1        36
##  3 2.01e7    14       11       21       8            5605      1        82
##  4 2.01e7    16       14       18      10            3529      1        52
##  5 2.01e7    12       10       12       7            2364      1        46
##   6 2.01e7     15        12        15       12                36       1          18
##   7 2.01e7     10         8        11        7               963       1          31
##   8 2.01e7     18        15        21       12              3005       1          39
##   9 2.01e7     16        15        16       13              1787       1          22
##  10 2.01e7     11        10        12        6               608       1          20
## # ... with 65 more rows, and 2 more variables: avgTimeOnPage <dbl>,
## #   userGender <fct>
```

5.

```r
analyticsRaw[analyticsRaw$sessions > 12 & analyticsRaw$sessions < 18,
             c("userGender", "date")]
```

```
## # A tibble: 44 x 2
##    userGender     date
##    <fct>         <int>
##  1 female     20140521
##  2 male       20140521
##  3 male       20140526
##  4 male       20140602
##  5 male       20140617
##  6 male       20140619
##  7 male       20140620
##  8 male       20140625
##  9 male       20140627
## 10 male       20140630
## # ... with 34 more rows
```

# 5 Data Management with `dplyr`

An alternative to the `base` approach in `R` to manage data is the package `dplyr`. It is the next iteration of `plyr` and especially useful for working with big data frames. The idea, as Hadley Wickham, one of the two authors of the package, states it, is: “Instead of moving the data to where the computation is, you want to send the computation to where the data is”. This concept enhances the speed of data manipulation fundamentally and makes calculations 100 up to 10000 times faster than under plyr. The concept of chaining or pipelining allows to write very clear-cut code which is read from left to right and top to bottom. Additionally, `dplyr` facilitates working with data from online databases.

The following section is mainly based on the package’s (shortened) introductory vignette[19] supplemented by examples from the Google Analytics dataset. The vignette is really worth reading. Further information about `dplyr` can be found in the official `dplyr` reference manual and several additional vignettes in CRAN (type `browseVignettes(package = "dplyr")` to see all of them). Also there is an online tutorial[20] and the webinar “Expressing yourself in R”[21] by Hadley Wickham. Another online tutorial by Kevin Markham gives a nice overview of `dplyr`[22]. Just to name a few…

[19] https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html
[20] https://www.dropbox.com/sh/i8qnluwmuieicxc/AAAgt9tIKoIm7WZKIyK25lh6a
[21] https://www.youtube.com/watch?v=wki0BqlztCo
[22] http://www.dataschool.io/dplyr-tutorial-for-faster-data-manipulation-in-r

## 5.1 General Structure

The code for managing data boils down to five basic functions: `arrange()`, `filter()`, `select()`, `mutate()`, and `summarise()`. They provide the basis of a language of data manipulation and are all very similar:

- The first argument of the function is a data frame.
- The subsequent arguments describe what to do with it, and you can refer to columns in the data frame directly without using `$` or quotes.
- The result or output is a new data frame.

Together these properties make it easy to chain multiple simple steps together to achieve a complex result.

Below, the following functions are presented:

- `filter()` and `slice()`
- `arrange()`
- `select()` and `rename()`
- `distinct()`
- `mutate()` and `transmute()`
- `summarise()`
- `sample_n()` and `sample_frac()`

In combination with `group_by()`, they cover the essential functionalities of data management. In the following we will show you how to use these `dplyr` functions with the help of the Google Analytics dataset.

```r
library(readr)
library(dplyr)
analytics <- read_csv(file = paste0(pathData, "Analytics.csv"))
```

*Note:* `dplyr` can work with data frames as is, but if you're dealing with large data, it's worthwhile to convert them to a `tbl_df`: this is a wrapper around a data frame that won't accidentally print a lot of data to the screen, which would take a lot of time.

## 5.2 `filter()` and `slice()`

The `filter()` function allows you to select a subset of rows from a dataset. First you need to specify the name of the dataset followed by the filtering expressions, whereby you can give as many filtering expressions as you like. Now try to select the rows from `analytics` where `userGender` is `male` and the `sessionDuration` is less than 300.

```
filter(analytics, userGender == "male" &
         sessionDuration < 300)
```

```
## # A tibble: 5 x 10
##       date users newUsers sessions bounces sessionDuration pageviews
##      <int> <int>    <int>    <int>   <int>           <int>     <int>
## 1 20140602    15       12       15      12              36        18
## 2 20140710    11        7       11       6             139        18
## 3 20140819    18       14       21      16             293        34
## 4 20140901    10       10       10       8             256        30
## 5 20140929    11        9       12       8             110        17
## # ... with 3 more variables: avgTimeOnPage <dbl>, userGender <chr>,
## #   bouncesHigh <chr>
```

This is equivalent to the more verbose `base` solution:

```
analytics[analytics$userGender == "male" &
            analytics$sessionDuration <= 300, ]
```

To select rows by position, use the `slice()` function:

```
slice(analytics, 10:12)
```

```
## # A tibble: 3 x 10
##       date users newUsers sessions bounces sessionDuration pageviews
##      <int> <int>    <int>    <int>   <int>           <int>     <int>
## 1 20140617    16       15       16      13            1787        22
## 2 20140618    11       10       12       6             608        20
## 3 20140619    14       14       14       8            1558        22
## # ... with 3 more variables: avgTimeOnPage <dbl>, userGender <chr>,
## #   bouncesHigh <chr>
```

The corresponding code in base is the following:

```
analytics[10:12, ]
```

## 5.3 arrange()

As the name suggests, the `arrange()` function orders the rows by the given condition(s). The default order is ascending. We'd like to reorder the analytics data by `sessionDuration`:

```
arrange(analytics, sessionDuration)
```

```
## # A tibble: 102 x 10
##        date users newUsers sessions bounces sessionDuration pageviews
##       <int> <int>    <int>    <int>   <int>           <int>     <int>
##  1 20140602    15       12       15      12              36        18
##  2 20140701    10        9       11       9             104        13
##  3 20140929    11        9       12       8             110        17
##  4 20140710    11        7       11       6             139        18
##  5 20140717    11       10       14      11             150        18
##  6 20140930    11       10       11       9             150        15
##  7 20140911    12        9       13       9             200        25
##  8 20140901    10       10       10       8             256        30
##  9 20140819    18       14       21      16             293        34
## 10 20140913    11       10       11       7             367        23
## # ... with 92 more rows, and 3 more variables: avgTimeOnPage <dbl>,
## #   userGender <chr>, bouncesHigh <chr>
```

If you'd like the analytics dataset to be ordered by descending `sessionDuration`, give:

```
arrange(analytics, desc(sessionDuration))
```

The previous two commands translated into `base`:

```
analytics[order(analytics$sessionDuration), ]
analytics[order(desc(analytics$sessionDuration)), ]
```

As it was the case with the `filter()` function, you can give multiple arguments to the `arrange()` function.

### 5.4 select() and rename()

The counterpart to the `filter()` function is `select()`, with which you can select a subset of *columns* from the dataset. You can simply specify the names of the desired columns:

```
select(analytics, date, users, pageviews)
```

```
## # A tibble: 102 x 3
##        date users pageviews
##       <int> <int>     <int>
##  1 20140521    12        18
##  2 20140521    13        26
##  3 20140526    12        36
##  4 20140527    14        82
##  5 20140528    16        52
##  6 20140530    12        46
##  7 20140602    15        18
##  8 20140606    10        31
##  9 20140611    18        39
## 10 20140617    16        22
## # ... with 92 more rows
```

This is fairly similar to the `subset` command in the base package:

```
subset(analytics, select = c(date, users, pageviews))
```

Another way to select certain columns in the `base` package is the following:

```
analytics[, c("date", "users", "pageviews")]
```

The `select()` function is also useful to omit certain columns from the dataset. For example, let's pretend we dont need the three columns `bounces`, `sessionDuration` and `pageviews`:

```
select(analytics, -c(bounces, sessionDuration, pageviews))
```

```
## # A tibble: 102 x 7
##        date users newUsers sessions avgTimeOnPage userGender bouncesHigh
##       <int> <int>    <int>    <int>         <dbl> <chr>      <chr>
##  1 20140521    12       11       13          85.2 female     >=10 bounces
##  2 20140521    13       10       14          58.5 male       >=10 bounces
##  3 20140526    12       10       15         124.  male       >=10 bounces
##  4 20140527    14       11       21          91.9 male       <10 bounces
##  5 20140528    16       14       18         104.  male       >=10 bounces
##  6 20140530    12       10       12          69.5 male       <10 bounces
##  7 20140602    15       12       15          12.0 male       >=10 bounces
##  8 20140606    10        8       11          48.1 male       <10 bounces
##  9 20140611    18       15       21         167.  male       >=10 bounces
## 10 20140617    16       15       16         298.  male       >=10 bounces
## # ... with 92 more rows
```

To omit certain columns from the dataset, the `subset()` command from the `base` package is again useful:

```
subset(analytics, select = -c(bounces, sessionDuration, pageviews))
```

There are a number of helper functions you can use within `select()`, like:

- `starts_with()`
- `ends_with()`
- `matches()`
- `contains()`

These let you quickly match larger blocks of variables that meet some criterion. See `?select` for more details.

`select()` can also be used to rename variables, but it drops the variables not mentioned.

```
select(analytics, dateYmd = date)
```

```
## # A tibble: 102 x 1
##     dateYmd
##       <int>
##   1 20140521
##   2 20140521
##   3 20140526
##   4 20140527
##   5 20140528
##   6 20140530
##   7 20140602
##   8 20140606
##   9 20140611
##  10 20140617
## # ... with 92 more rows
```

So if you want to keep all variables but just rename a few, the `rename()` function is the better choice:

```
rename(analytics, dateYmd = date)
```

```
## # A tibble: 102 x 10
##     dateYmd users newUsers sessions bounces sessionDuration pageviews
##       <int> <int>    <int>    <int>   <int>           <int>     <int>
##   1 20140521    12       11       13      10             425        18
##   2 20140521    13       10       14      10             703        26
##   3 20140526    12       10       15      10            2596        36
##   4 20140527    14       11       21       8            5605        82
##   5 20140528    16       14       18      10            3529        52
##   6 20140530    12       10       12       7            2364        46
##   7 20140602    15       12       15      12              36        18
##   8 20140606    10        8       11       7             963        31
##   9 20140611    18       15       21      12            3005        39
##  10 20140617    16       15       16      13            1787        22
## # ... with 92 more rows, and 3 more variables: avgTimeOnPage <dbl>,
## #   userGender <chr>, bouncesHigh <chr>
```

In the `base` package, this is somewhat less straightforward:

```
names(analytics)[names(analytics) == "date"] <- "dateYmd"
```

The function `distinct()` corresponds to the `unique()` function from the `base` package but should be faster. It helps to extract distinct (unique) rows in a dataset.

```
distinct(select(analytics, newUsers, userGender))
```

```
## # A tibble: 23 x 2
##    newUsers userGender
##       <int> <chr>
##  1       11 female
##  2       10 male
##  3       11 male
##  4       14 male
##  5       12 male
##  6        8 male
##  7       15 male
##  8        9 male
##  9        8 female
## 10       18 male
## # ... with 13 more rows
```

## 5.5 `mutate()` and `transmute()`

The function `mutate()` is used to add new columns that are functions of columns which already exist in your dataset. Now we will add a column called `avgSessionDuration` which gives the average session duration as the ratio of `sessionDuration` and `sessions`. For a better overview, we select only some columns:

```
analytics <- mutate(analytics, avgSessionDuration = sessionDuration / sessions)
select(analytics, sessionDuration, sessions, avgSessionDuration)
```

```
## # A tibble: 102 x 3
##    sessionDuration sessions avgSessionDuration
##              <int>    <int>              <dbl>
##  1             425       13               32.7
##  2             703       14               50.2
##  3            2596       15              173.
##  4            5605       21              267.
##  5            3529       18              196.
##  6            2364       12              197.
##  7              36       15                2.40
##  8             963       11               87.5
##  9            3005       21              143.
## 10            1787       16              112.
## # ... with 92 more rows
```

`mutate()` basically does the same as the base command `transform()` but additionally allows you to refer to columns that you just created. If you'd only like to keep the new columns and drop the rest, use the `transmute()` function instead.

## 5.6 summarise()

And last but not least: the `summarise()` function. It returns a single row by collapsing many values to a summary specified by your argument(s). In the following example the average number of users is calculated:

```
summarise(analytics, averageNoUsers = mean(users))
```

```
## # A tibble: 1 x 1
##   averageNoUsers
##            <dbl>
## 1           14.1
```

In many cases the `summarise()` function is not really useful since the same summary statistic can be obtained by applying the function directly to the variable:

```
mean(analytics$users)
```

```
## [1] 14.08824
```

But it becomes more powerful when combined with `group_by` – you will see that in section 5.8.

## 5.7 `sample_n()` and `sample_frac()`

And finally there are two functions to draw a random sample of rows: `sample_n()` for taking a fixed number and `sample_frac()` for taking a fixed fraction.

For example, we could extract four random rows from the analytics dataset with:

```
sample_n(analytics, size = 4)
```

```
## # A tibble: 4 x 11
##    dateYmd users newUsers sessions bounces sessionDuration pageviews
##      <int> <int>    <int>    <int>   <int>           <int>     <int>
## 1 20140821    13       11       15      11            7532       106
## 2 20140909    19       12       19      11            1381        42
## 3 20140623    21       18       23      16            5247       110
## 4 20140707    16       14       16      12             598        37
## # ... with 4 more variables: avgTimeOnPage <dbl>, userGender <chr>,
## #   bouncesHigh <chr>, avgSessionDuration <dbl>
```

For a sample of 5% of the datasets rows we'd type:

```
sample_frac(analytics, size = 0.05)
```

### 5.8 group_by()

The five functions we introduced in this section become really powerful when you use them in combination with the `group_by()` function. It is used prior to the `dplyr` functions and the resulting data frame must be assigned to a new object.

```
analyticsGender <- group_by(analytics, userGender)
summarise(analyticsGender, avgNoUsers = mean(users),
          avgPageViews = mean(pageviews))
```

```
## # A tibble: 2 x 3
##   userGender avgNoUsers avgPageViews
##   <chr>           <dbl>        <dbl>
## 1 female           13.3         25.9
## 2 male             14.4         44.6
```

The functions are affected by grouping as follows:

- Grouped `select()` is the same as ungrouped `select()`, except that grouping variables are always retained.
- Grouped `arrange()` orders first by grouping variable(s) and inside of the groups by the variable(s) you specify.
- `mutate()` and `filter()` are most useful in conjunction with window functions (like `rank()`, or `min(x) == x`), and are described in detail in `vignette("window-function")`. Variables computed with mutate are computed within each group.
- `sample_n()` and `sample_frac()` sample the specified number/fraction of rows in each group.
- `slice()` extracts rows within each group.
- `summarise()` is easy to understand and very useful, use it with aggregate functions, which take a vector of values, and return a single number. The result is computed for each group separately.

The following example shows the interplay of `mutate` and `summarise` with `group_by`:

```
# Ungrouped: summarise computes overall mean
summarise(analytics, avgSessionDuration = mean(sessionDuration))
```

```
## # A tibble: 1 x 1
##   avgSessionDuration
##                <dbl>
## 1              2541.
```

```

# Grouped: summarise computes mean for each group
summarise(analyticsGender, avgSessionDuration = mean(sessionDuration))
```

```
## # A tibble: 2 x 2
##   userGender avgSessionDuration
##   <chr>                   <dbl>
## 1 female                  1623.
## 2 male                    2872.
```

```r
# Ungrouped: mean session duration appears in each row of a new colum
mutate(analytics, avgSessionDuration = mean(sessionDuration))[, - (1:7)]
```

```
## # A tibble: 102 x 4
##    avgTimeOnPage userGender bouncesHigh  avgSessionDuration
##            <dbl> <chr>      <chr>                     <dbl>
##  1          85.2 female     >=10 bounces              2541.
##  2          58.5 male       >=10 bounces              2541.
##  3         124.  male       >=10 bounces              2541.
##  4          91.9 male       <10 bounces               2541.
##  5         104.  male       >=10 bounces              2541.
##  6          69.5 male       <10 bounces               2541.
##  7          12.0 male       >=10 bounces              2541.
##  8          48.1 male       <10 bounces               2541.
##  9         167.  male       >=10 bounces              2541.
## 10         298.  male       >=10 bounces              2541.
## # ... with 92 more rows
```

```r
# Grouped: mean session duration of the respective user gender appears in each
    row of a new colum:
mutate(analyticsGender, avgSessionDuration = mean(sessionDuration))[, - (1:7)]
```

```
## # A tibble: 102 x 4
## # Groups:   userGender [2]
##    avgTimeOnPage userGender bouncesHigh  avgSessionDuration
##            <dbl> <chr>      <chr>                     <dbl>
##  1          85.2 female     >=10 bounces              1623.
##  2          58.5 male       >=10 bounces              2872.
##  3         124.  male       >=10 bounces              2872.
##  4          91.9 male       <10 bounces               2872.
##  5         104.  male       >=10 bounces              2872.
##  6          69.5 male       <10 bounces               2872.
##  7          12.0 male       >=10 bounces              2872.
##  8          48.1 male       <10 bounces               2872.
##  9         167.  male       >=10 bounces              2872.
## 10         298.  male       >=10 bounces              2872.
## # ... with 92 more rows
```

## 5.9 Chaining

One advantage of `dplyr` is that it is more readable than general `base` code. This is mainly because of the concept of “chaining” or “pipelining”, i.e., the possibility to write commands in a natural order connected by the infix operator `%>%`. So we can reuse the `group_by()` example from above but do not have to assign the grouped dataset:

```
analytics %>%
  group_by(userGender) %>%
  summarise(avgNoUsers = mean(users),
            avgPageViews = mean(pageviews))
```

```
## # A tibble: 2 x 3
##   userGender avgNoUsers avgPageViews
##   <chr>           <dbl>        <dbl>
## 1 female           13.3         25.9
## 2 male             14.4         44.6
```

Chaining can prevent not only the confusion of nested functions but can also save quite some lines of code.

Another example: If you combine `slice` and `arrange`, you can easily extract the rows with the highest (or lowest) values on a variable. With chaining, the code is much easier to read:

```
# Without chaining
slice(arrange(analytics, sessionDuration), 1:5)
```

```
## # A tibble: 5 x 11
##    dateYmd users newUsers sessions bounces sessionDuration pageviews
##      <int> <int>    <int>    <int>   <int>           <int>     <int>
## 1 20140602    15       12       15      12              36        18
## 2 20140701    10        9       11       9             104        13
## 3 20140929    11        9       12       8             110        17
## 4 20140710    11        7       11       6             139        18
## 5 20140717    11       10       14      11             150        18
## # ... with 4 more variables: avgTimeOnPage <dbl>, userGender <chr>,
## #   bouncesHigh <chr>, avgSessionDuration <dbl>
```

```

# With chaining
analytics %>% arrange(desc(sessionDuration)) %>% slice(1:5)
```

```
## # A tibble: 5 x 11
##    dateYmd users newUsers sessions bounces sessionDuration pageviews
##      <int> <int>    <int>    <int>   <int>           <int>     <int>
## 1 20140820    16       13       19      11           18964       207
## 2 20140818    11        7       13       9            9809       115
## 3 20140910    17       13       22      13            9147        73
## 4 20140825    17       14       19      11            7805        76
## 5 20140911    13       12       16      11            7630        88
## # ... with 4 more variables: avgTimeOnPage <dbl>, userGender <chr>,
## #   bouncesHigh <chr>, avgSessionDuration <dbl>
```

## 5.10 dplyr and Databases

Another convenience of the dplyr package lies in the possibility to work with data either stored in data tables or in databases. In each case the syntax remains the same. When working with a database it is however only possible to use SELECT statements. So far SQLite, PostgreSQL/Redshift, MySQL/MariaDB, BigQuery and MonetDB are supported.

For simple queries dplyr is well-suited. Internally it translates R code into SQL. This is especially appealing because translation is available into different SQL dialects. Since this translation does not always produce the most performing code, you are well-adviced to write statements directly in SQL if queries become more complex and time-consuming (see Section 3.2).

The database backend for dplyr is provided by the dbplyr package. In general you won't need to use any function from dbplyr directly, just make sure that the package has been installed. For more information see https://cran.r-project.org/web/packages/dbplyr/vignettes/dbplyr.html.

### Further Reading

- Wickham, H., & Grolemund, G. (2017). R for Data Science. Available under r4ds.had.co.nz/
- Introduction to dplyr. dplyr vignette, available under cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html

**Exercises**

In the last section, we showed you how to make use of Hadley Wickham's package `dplyr`. Now you get the chance to work with `dplyr` by yourself. We prepared some exercises for the famous `titanic` dataset. The solutions can be found at the end of this section. So load the dataset and take a look at it. Additionally, do not forget to load the required packages `dplyr` and `readr`.

```
library(dplyr)
library(readr)
titanic <- read_csv(paste0(pathData, "titanic.csv"))
## Warning: Missing column names filled in: 'X1' [1]

head(titanic)
```

```
## # A tibble: 6 x 15
##      X1 pclass survived name   sex      age sibsp parch ticket  fare cabin
##   <int> <chr>     <int> <chr>  <chr>  <dbl> <int> <int> <chr>  <dbl> <chr>
## 1     1 1st           1 Allen~ fema~ 29.0       0     0 24160  211.  B5
## 2     2 1st           1 Allis~ male   0.917     1     2 113781 152.  C22 ~
## 3     3 1st           0 Allis~ fema~  2.00      1     2 113781 152.  C22 ~
## 4     4 1st           0 Allis~ male  30.0       1     2 113781 152.  C22 ~
## 5     5 1st           0 Allis~ fema~ 25.0       1     2 113781 152.  C22 ~
## 6     6 1st           1 Ander~ male  48.0       0     0 19952   26.5 E12
## # ... with 4 more variables: embarked <chr>, boat <chr>, body <int>,
## #   home.dest <chr>
```

| Variable | Description |
|---|---|
| `pclass` | passenger class (1st, 2nd, 3rd) |
| `survived` | yes (1), no (0) |
| `name` | the passenger's name |
| `sex` | female, male |
| `age` | age in years |
| `sibsp` | number of siblings/spouses aboard |
| `parch` | number parents/children aboard |
| `ticket` | the ticket number |
| `fare` | passenger fare in pounds |
| `cabin` | the passenger's cabin |
| `embarked` | port of embarkation (Cherbourg, Queenstown or Southampton) |
| `boat` | lifeboat |
| `body` | body identification number |
| `home.dest` | home/destination |

Now it's your turn! If you don't know the answer right away, don't be afraid to try around. Producing errors cannot do any harm – it just makes you learn and lets you develop creative solutions.

1. The first column seems to contain only the row number which we won't need for our analysis. So go ahead and delete it right away.
2. When talking about the Titanic, what is most important, besides how Leo met Kate, is who survived.
   a) Create a table, listing for each passenger class the proportion of people who survived.
   b) It looks like you just had to travel first class in order to survive. But did this hold for everybody? Redo the previous table for men and women separately. What do you notice?
3. Ok, back to the original dataset.
   a) Omit the variables `sibsp`, `parch`, and `ticket`. Save the resulting data frame in a new dataset.

b) Now print the mean age of the Titanic passengers grouped by the variable `survived` (careful, the `age` column contains `NA`s!). Who was more likely to survive? Was this true for men as well as for women?

4. Use the dataset from the task before, arrange it by age and look at the five oldest people aboard. Rearrange the dataset and look at the five youngest people aboard.

5. Let's check whether the age of the young people of the previous task could be a coding error. So use the variable `parch` to check whether the respective person has parents on board. If so, we could assume that the age is valid and we are looking at little children.

6. Well, it could also be the case that young children are accompanied by their siblings.

   a) Create a new column called "alone" that indicates if the person was alone aboard (no siblings/spouses and no parents/children) or not.
   b) Look at the minimum age in the group of people travelling alone. That's still pretty young, isn't it?
   c) Now select those rows of people who are travelling alone and who are younger than the 25% age quantile. How high is the percentage of survivors in this group?

7. So let's see whether or not there is an association between the fare people paid and their survival.

   a) Group the dataset according to whether or not they survived and calculate the $1^{st}$ quantile, the mean, the median, and the $3^{rd}$ quantile of the fare for both groups (name the columns of the resulting table appropriately). Check if the variable `fare` contains any `NA`s first (hint: use the function `is.na()`).
   b) It looks like one had to pay a higher fare in order to survive. But in the beginning, we saw that the chance to survive was higher in the $1^{st}$ compared to the $2^{nd}$ class and higher in the $2^{nd}$ compared to the $3^{rd}$ class. Recalculate the statistics about the fare controlling for the passenger class. Does the previously found relationship between ticket price and survival still hold?

8. To end this sad chapter about the sinking of the Titanic, let's do some selection exercises. Always save the resulting data frame in a new object.

   a) Look at the passengers who had at least more than 2 siblings/spouses or more than 2 parents/children aboard.
   b) Among these passengers, select the rows of passengers who travelled $3^{rd}$ class and whose age is 18 or lower.
   c) Next, we'd like to have only the rows from the remaining dataset where the variable `body` is not missing.
   d) And finally, select the rows for passengers whose name contains "Henry" (hint: use `grepl`).

## Solutions

1.

```
# Remove first column from dataset
titanic <- titanic %>% select(-1)
```

2.

a)

```
# Table of survival by passenger class
titanic %>% group_by(pclass) %>% summarise(propSurvived = mean(survived))
```

```
## # A tibble: 3 x 2
##   pclass propSurvived
##   <chr>         <dbl>
## 1 1st           0.619
## 2 2nd           0.430
## 3 3rd           0.255
```

b)

```
# Table of survival by passenger class and sex
titanic %>% group_by(sex, pclass) %>% summarise(propSurvived = mean(survived))
```

```
## # A tibble: 6 x 3
## # Groups:   sex [?]
##   sex    pclass propSurvived
##   <chr>  <chr>         <dbl>
## 1 female 1st           0.965
## 2 female 2nd           0.887
## 3 female 3rd           0.491
## 4 male   1st           0.341
## 5 male   2nd           0.146
## 6 male   3rd           0.152
```

An alternative way to compute the proportion of surviving people would be `propSurvived = sum(survived == 1) / n()`, where `n()` returns the number of rows in the respective group.

3.

a)

```
titanicSel <- titanic %>% select(-c(sibsp, parch, ticket))
```

b)

```
# Mean age of surviving people
titanicSel %>%
  group_by(survived) %>%
  summarise(meanAge = mean(age, na.rm = TRUE))
```

```
## # A tibble: 2 x 2
##   survived meanAge
##      <int>   <dbl>
## 1        0    30.5
## 2        1    28.9
```

```
# Mean age of surviving grouped by gender
titanicSel %>%
  group_by(sex, survived) %>%
  summarise(meanAge = mean(age, na.rm = TRUE))
```

```
## # A tibble: 4 x 3
## # Groups:   sex [?]
##   sex    survived meanAge
##   <chr>     <int>   <dbl>
## 1 female        0    25.3
## 2 female        1    29.8
## 3 male          0    31.5
## 4 male          1    27.0
```

4.

```
# Looking at oldest and youngest passengers
titanicSel %>% arrange(desc(age)) %>% slice(1:5)
```

```
## # A tibble: 5 x 11
##   pclass survived name        sex     age  fare cabin embarked boat   body
##   <chr>     <int> <chr>       <chr> <dbl> <dbl> <chr> <chr>    <chr> <int>
## 1 1st           1 Barkworth,~ male    80.  30.0  A23   Southam~ B        NA
## 2 1st           1 Cavendish,~ fema~   76.  78.8  C46   Southam~ 6        NA
## 3 3rd           0 Svensson, ~ male    74.   7.78 <NA>  Southam~ <NA>     NA
## 4 1st           0 Artagaveyt~ male    71.  49.5  <NA>  Cherbou~ <NA>     22
## 5 1st           0 Goldschmid~ male    71.  34.7  A5    Cherbou~ <NA>     NA
## # ... with 1 more variable: home.dest <chr>
```

```

titanicSel %>% arrange(age) %>% slice(1:5)
```

```
## # A tibble: 5 x 11
##   pclass survived name        sex     age  fare cabin embarked boat   body
##   <chr>     <int> <chr>       <chr> <dbl> <dbl> <chr> <chr>    <chr> <int>
## 1 3rd            1 "Dean, Mis~ fema~ 0.167 20.6  <NA>  Southam~ 10       NA
## 2 3rd            0 Danbom, Ma~ male  0.333 14.4  <NA>  Southam~ <NA>     NA
## 3 3rd            1 Thomas, Ma~ male  0.417  8.52 <NA>  Cherbou~ 16       NA
## 4 2nd            1 Hamalainen~ male  0.667 14.5  <NA>  Southam~ 4        NA
## 5 3rd            1 Baclini, M~ fema~ 0.750 19.3  <NA>  Cherbou~ C        NA
## # ... with 1 more variable: home.dest <chr>
```

5.

```r
# Checking parch for young people
titanic %>%
  select(age, parch, sex) %>%
  arrange(age)
```

```
## # A tibble: 1,309 x 3
##      age parch sex
##    <dbl> <int> <chr>
##  1 0.167     2 female
##  2 0.333     2 male
##  3 0.417     1 male
##  4 0.667     1 male
##  5 0.750     1 female
##  6 0.750     1 female
##  7 0.750     1 male
##  8 0.833     2 male
##  9 0.833     1 male
## 10 0.833     1 male
## # ... with 1,299 more rows
```

6.

There are several ways to solve this exercise. To give you an idea about R's flexibility, we present some of them. For the example at hand it doesn't really matter which way to chose but when dealing with big datasets, performance will matter and thus one of the solutions can become superior to the others.

a)

```r
# Generate new variable indicating whether person was alone or not
titanic <- titanic %>%
  mutate(alone = as.numeric(sibsp == 0 & parch == 0))
```

b)

```r
titanic %>%
  group_by(alone) %>%
  summarise(minAge = min(age, na.rm = TRUE))
```

```
## # A tibble: 2 x 2
##   alone minAge
##   <dbl>  <dbl>
## 1    0.  0.167
## 2    1.  5.00
```

c)

```
# Filter out persons who where alone and younger than 25% quantile
titanic %>%
  filter(alone == TRUE & age < quantile(age, p = 0.25, na.rm = TRUE)) %>%
  arrange(age) %>%
  summarise(propSurvived = mean(survived, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##   propSurvived
##          <dbl>
## 1        0.305
```

7.

a)

```
sum(is.na(titanic$fare))
```

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

```
# Statistics about fare by survival
titanic %>% group_by(survived) %>%
    summarize(fare1st = quantile(fare, na.rm = TRUE, p = 0.25),
              fareMean = mean(fare, na.rm = TRUE),
              fareMed = median(fare, na.rm = TRUE),
              fare3rd = quantile(fare, na.rm = TRUE, p = 0.75))
```

```
## # A tibble: 2 x 5
##   survived fare1st fareMean fareMed fare3rd
##      <int>   <dbl>    <dbl>   <dbl>   <dbl>
## 1        0    7.85     23.4    10.5    26.0
## 2        1   11.2      49.4    26.0    57.8
```

b)

```
# Statistics about fare in survival groups controlling for passenger class
titanic %>% group_by(pclass, survived) %>%
    summarize(fare1st = quantile(fare, na.rm = TRUE, p = 0.25),
              fareMean = mean(fare, na.rm = TRUE),
              fareMed = median(fare, na.rm = TRUE),
              fare3rd = quantile(fare, na.rm = TRUE, p = 0.75))
```

```
## # A tibble: 6 x 6
## # Groups:   pclass [?]
##   pclass survived fare1st fareMean fareMed fare3rd
##   <chr>     <int>   <dbl>    <dbl>   <dbl>   <dbl>
## 1 1st           0   27.7      70.4   50.0     79.4
## 2 1st           1   49.5      98.0   76.7    120.
## 3 2nd           0   12.4      19.8   13.0     26.0
## 4 2nd           1   13.0      23.0   23.0     27.7
## 5 3rd           0    7.75     13.4    8.05    14.5
## 6 3rd           1    7.75     12.9    8.05    15.7
```

8.

a)

```
titanicFilt <- titanic %>%
  filter(sibsp > 2 | parch > 2)
```

b)

```
titanicFilt <- titanicFilt %>%
  filter(pclass == "3rd" & age <= 18)
```

c)

```
titanicFilt <- titanicFilt %>%
  filter(!is.na(body))
```

d)

```
titanicFilt <- titanicFilt %>%
  filter(grepl("Henry", name))

titanicFilt
```

We can also do this in one chunk of code:

```
titanicFilt <- titanic %>%
  filter((sibsp > 2 | parch > 2) &
           pclass == "3rd" & age <= 18 &
           !is.na(body) &
           grepl("Henry", name))
```

# 6 Descriptive Statistics

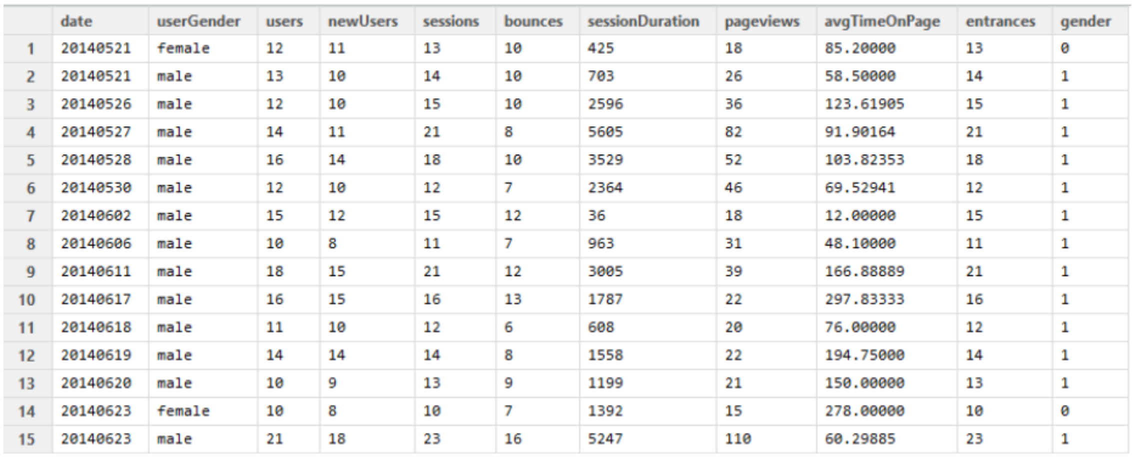
"Each and every investigation should start with an explorative data analysis" (Tukey 1977).

Many researchers ignore this step and start off with inferential statistics. But then they lack a "feeling" for their data, potential errors introduced through transcription or coding remain undetected and standard assumptions (like a normal distribution of the data) are not challenged. Well, you can choose a different path. A short descriptive analysis including some graphics already helps to "get to know" your data and to improve your analysis.

As we will work with data files in this course, we first save the path to our data folder in an R object so we don't have to type it again and again:

```r
pathData <- "PATH_TO_YOUR_DESKTOP/yourDataFolder"
```

We will work with a Google Analytics dataset[23] (see Figure 6). The observations are identified by the fixed columns `date` and `userGender` which come first. Then the different measured variables follow. Each row holds an observation and each column one variable. It contains tracking data about the behavior of visitors of the INWT Statistics website. Observations are grouped by date and gender, hence, each row contains a unique date-gender-combination. So the first line of the data can be read as follows: On May $21^{st}$ in 2014, 12 female users visited the INWT website. Among them were 11 new users, i.e., their cookie has not been recognized. All in all they started 13 sessions of which there were 10 bounces. This means that they stayed only on one page, leaving it unclear how long they stayed or where they went afterwards. The 13 sessions lasted a total of 425 seconds. There have been 18 pageviews with an average time on page of 85.2 seconds. The variable `entrances` contains the same information as the variable `sessions`. Same holds for the variable `gender` which is, except for the coding, the same as `userGender`.



| | date | userGender | users | newUsers | sessions | bounces | sessionDuration | pageviews | avgTimeOnPage | entrances | gender |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 20140521 | female | 12 | 11 | 13 | 10 | 425 | 18 | 85.20000 | 13 | 0 |
| 2 | 20140521 | male | 13 | 10 | 14 | 10 | 703 | 26 | 58.50000 | 14 | 1 |
| 3 | 20140526 | male | 12 | 10 | 15 | 10 | 2596 | 36 | 123.61905 | 15 | 1 |
| 4 | 20140527 | male | 14 | 11 | 21 | 8 | 5605 | 82 | 91.90164 | 21 | 1 |
| 5 | 20140528 | male | 16 | 14 | 18 | 10 | 3529 | 52 | 103.82353 | 18 | 1 |
| 6 | 20140530 | male | 12 | 10 | 12 | 7 | 2364 | 46 | 69.52941 | 12 | 1 |
| 7 | 20140602 | male | 15 | 12 | 15 | 12 | 36 | 18 | 12.00000 | 15 | 1 |
| 8 | 20140606 | male | 10 | 8 | 11 | 7 | 963 | 31 | 48.10000 | 11 | 1 |
| 9 | 20140611 | male | 18 | 15 | 21 | 12 | 3005 | 39 | 166.88889 | 21 | 1 |
| 10 | 20140617 | male | 16 | 15 | 16 | 13 | 1787 | 22 | 297.83333 | 16 | 1 |
| 11 | 20140618 | male | 11 | 10 | 12 | 6 | 608 | 20 | 76.00000 | 12 | 1 |
| 12 | 20140619 | male | 14 | 14 | 14 | 8 | 1558 | 22 | 194.75000 | 14 | 1 |
| 13 | 20140620 | male | 10 | 9 | 13 | 9 | 1199 | 21 | 150.00000 | 13 | 1 |
| 14 | 20140623 | female | 10 | 8 | 10 | 7 | 1392 | 15 | 278.00000 | 10 | 0 |
| 15 | 20140623 | male | 21 | 18 | 23 | 16 | 5247 | 110 | 60.29885 | 23 | 1 |

**Figure 6** – A subset from the Google Analytics dataset

A description of the data could take the following form and is fairly helpful for analysis:

[23]Note that in order to enhance readability of the commands to be presented below, results have not always been rounded where necessary. Please remember, however, that most of the time, the $4^{th}$ decimal place is not really important.

| Variable | Description | Measurement scale | Values |
|---|---|---|---|
| `date` | date of access | metric | |
| `userGender` | gender of user | nominal | female, male |
| `users` | number of users | metric | |
| `newUsers` | number of new users | metric | |
| `sessions` | number of sessions | metric | |
| `bounces` | number of bounces | metric | |
| `sessionDuration` | total duration of user sessions (seconds) | metric | |
| `pageviews` | total number of pageviews | metric | |
| `avTimeOnPage` | average time on page (seconds) | metric | |
| `gender` | gender of user | nominal | 0 (female), 1 (male) |

Note the definition of the variable `gender`, which holds the values 0 (for female) and 1 (for male).

```
library(readxl)
analytics <- read_excel(path = paste0(pathData, "analytics.xlsx"),
                        sheet = 1)
names(analytics)
```

```
##  [1] "date"            "userGender"      "users"
##  [4] "newUsers"        "sessions"        "bounces"
##  [7] "sessionDuration" "pageviews"       "avgTimeOnPage"
## [10] "entrances"
```

## 6.1 Basic statistics

Now, we want to calculate some "basic" statistics for the average time on page. The according functions in R are mainly self-explaining:

| Command | Output |
|---|---|
| mean() | Mean |
| median() | Median |
| sd() | Standard deviation |
| var() | Variance |
| cor() | Correlation coefficient |
| cov() | Covariance |
| min() | Minimum |
| max() | Maximum |
| quantile() | Quantile(s) |
| mad() | Median Absolute Deviation |

Some examples in this sections can be written in a more elegant way using the `dplyr` package. In these cases, we will present both solutions. If you are not yet familiar with `dplyr`, you can just skip the `dplyr` version for now.

So we start with mean, median, minimum and maximum[24].

```
mean(analytics$avgTimeOnPage)
```

```
## [1] 120.1263
```

```
median(analytics$avgTimeOnPage)
```

```
## [1] 102.3612
```

```
min(analytics$avgTimeOnPage)
```

```
## [1] 12
```

```
max(analytics$avgTimeOnPage)
```

```
## [1] 384.4
```

Quantiles can be determined with aid of the `quantile()` function:

[24]Note that of course these statistics and many more can be calculated using the 'dplyr ' package, too: `analytics %>% summarise(mean(avgTimeOnPage))`. This is, however, more complicated than the `base` solution.

```r
quantile(analytics$avgTimeOnPage, 0.5)
```

```
##      50%
## 102.3612
```

```r
quantile(analytics$avgTimeOnPage, c(0.25, 0.75))
```

```
##       25%       75%
##  60.13993 165.58333
```

```r
round(quantile(analytics$avgTimeOnPage, seq(0.1, 1, by = 0.1)), digits = 3)
```

```
##     10%     20%     30%     40%     50%     60%     70%     80%     90%
##  36.400  54.361  66.281  81.095 102.361 123.451 150.074 182.427 214.388
##    100%
## 384.400
```

The last command includes the `seq()` function in order to generate a sequence of values. This sequence starts at 0.1 (first argument), ends at 1 (second argument) and increases by steps of 0.1 (argument `by`). Additionally, the `round()` function is used to round the quantiles to up to 3 decimal places (`digits` = 3). Measures of dispersion are calculated analoguously:

```r
sd(analytics$avgTimeOnPage)
```

```
## [1] 78.73725
```

```r
var(analytics$avgTimeOnPage)
```

```
## [1] 6199.554
```

```r
mad(analytics$avgTimeOnPage)
```

```
## [1] 71.1467
```

```r
IQR(analytics$avgTimeOnPage)
```

```
## [1] 105.4434
```

```r
max(analytics$avgTimeOnPage) - min(analytics$avgTimeOnPage)
```

```
## [1] 372.4
```

The `summary()` function outputs a whole set of statistics for a vector:

```r
summary(analytics$avgTimeOnPage)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   12.00   60.14  102.36  120.13  165.58  384.40
```

The `table()` function generates a basic frequency table. In this example we count the men and women included in our sample:

```
table(analytics$userGender)
```

```
##
## female   male
##     27     75
```

The following code creates a new variable `bouncesHigh` in which the observations are grouped into two categories: one category with an amount of bounces lower than the median and another one with an amount of bounces greater or equal to the median:

```
library(dplyr)
median(analytics$bounces)
```

```
## [1] 10
```

```
analytics <- analytics %>% mutate(bouncesHigh = bounces >= median(bounces))
class(analytics$bouncesHigh)
```

```
## [1] "logical"
```

```
head(analytics$bouncesHigh)
```

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

```
analytics$bouncesHigh <- factor(analytics$bouncesHigh)
levels(analytics$bouncesHigh) <- c("<10 bounces", ">=10 bounces")
head(analytics$bouncesHigh)
```

```
## [1] >=10 bounces >=10 bounces >=10 bounces <10 bounces  >=10 bounces
## [6] <10 bounces
## Levels: <10 bounces >=10 bounces
```

The `table()` function can also be used to build contingency tables. In the following we distinguish between women and men as well as an amount of bounces lower than 10 and greater or equal to 10:

```
table(analytics$bouncesHigh, analytics$userGender)
```

```
##
##                 female male
##   <10 bounces        9   35
##   >=10 bounces      18   40
```

The contingency table can easily be supplemented with marginal totals:

```
addmargins(table(analytics$bouncesHigh, analytics$userGender))
```

```
##
##                 female male Sum
##   <10 bounces        9   35  44
##   >=10 bounces      18   40  58
##   Sum               27   75 102
```

Following the same logic, relative frequencies can be listed instead of absolute frequencies:

```
round(prop.table(table(analytics$bouncesHigh, analytics$userGender)), 2)
```

```
##
##                  female male
##   <10 bounces      0.09 0.34
##   >=10 bounces     0.18 0.39
```

Note that for both, `addmargins()` and `prop.table()`, the corresponding command is applied to a table object. Okay, but what if we want to know if women or men bounce more?

```
round(prop.table(table(analytics$bouncesHigh, analytics$userGender),
                 margin = 2),
      2)
```

```
##
##                  female male
##   <10 bounces      0.33 0.47
##   >=10 bounces     0.67 0.53
```

The parameter `margin` indicates whether percentages are calculated row-wise (1) or column-wise (2). In this case we calculate percentages column-wise. Now we can see that about 67% of the women have more than or equal to 10 bounces and about 33% less than 10. Men, however, exhibit a rather balanced bouncing behavior (53% vs. 47%).

With `dplyr` it's easy to show absolute frequencies, relative frequencies and relative frequencies with respect to one of the variables in the same table:

```
analytics %>%
  group_by(userGender, bouncesHigh) %>%
  summarise(Anzahl = n()) %>%
  ungroup %>%
  mutate(Anteil_gesamt = round(Anzahl / sum(Anzahl), 2)) %>%
  group_by(userGender) %>%
  mutate(Anteil_innerhalb_Geschlecht = round(Anzahl / sum(Anzahl), 2))
```

```
## # A tibble: 4 x 5
## # Groups:   userGender [2]
##   userGender bouncesHigh  Anzahl Anteil_gesamt Anteil_innerhalb_Geschlecht
##   <chr>      <fct>         <int>         <dbl>                       <dbl>
## 1 female     <10 bounces       9        0.0900                       0.330
## 2 female     >=10 bounces     18        0.180                        0.670
## 3 male       <10 bounces      35        0.340                        0.470
## 4 male       >=10 bounces     40        0.390                        0.530
```

In order to quantify interdependence between metric or ordinal variables, the correlation can be calculated. This is achieved using the `cor()` function in R. By specifying the optional parameter `method`, it is possible to choose between `pearson` (Pearson's correlation, metric scale of measurement, default), `kendall` (Kendall's rank correlation, at least ordinal scale of measurement) and `spearman` (Spearman's rank correlation, at least ordinal scale of measurement):

```r
cor(analytics$newUsers, analytics$bounces)
```

```
## [1] 0.7810729
```

```r
cor(analytics$newUsers, analytics$bounces, method = "spearman")
```

```
## [1] 0.7496159
```

*Note:* It is also possible to apply the `cor()` function to a whole dataset or matrix. The output consists of a correlation matrix in which the correlation of each variable with all the other variables is displayed.

The following two sections about the presentation of numbers in graphics and tables are very basic. They contain, however, some essential rules we find important to tell you about. We have produced and seen many presentations and want to share some lessons we have painfully learned. . .

### 6.2 Presenting numbers in graphics

What can generally be said about a picture? Well: "A picture is worth a thousand words!" (see Figure 7).

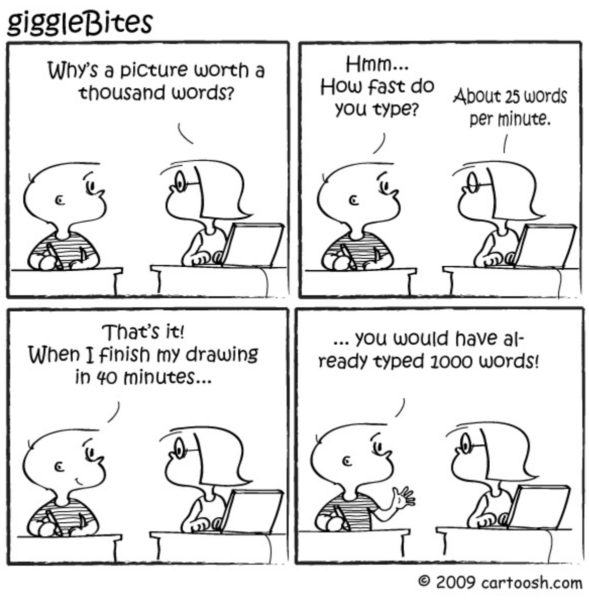

gigglebites
Why's a picture worth a thousand words?
Hmm... How fast do you type?
About 25 words per minute.
That's it! When I finish my drawing in 40 minutes...
... you would have already typed 1000 words!
© 2009 cartoosh.com

**Figure 7** – A Picture Worth a Thousand Words

Most readers do not really want to read, they rather want to look at a graphic in order to get the author's point. Hence, what is important for data scientists is to deliver the story of a graph at a glance. In the following, some graphs are introduced, accompanied by Dos and Dont's in order to help you to tell the story of your data.

#### 6.2.1 Static graphics with `ggplot2`

We are using the `R` package `ggplot2`, which provides a graphic language based on the so-called layered grammar of graphics. It combines the good parts of `graphics` and `lattice` graphics while avoiding their shortcomings. More information can be found on www.ggplot2.org. You can also find a link to Hadley Wickham's book there, which describes the theory underlying ggplot2: "ggplot2: Elegant Graphics for Data Analysis". The following explanations are mainly based on this book.

Essentially, the layered grammar of graphics consists of the following components:

- **data**, and a set of **aesthetic mappings** (visual properties): Every plot is based on a data frame. Aesthetics indicate how the data is mapped to aesthetic attributes, i.e., to elements we can see Examples are: colour, size, shape, transparency.

- **geoms** (geometric objects) are the graphical representation of a statistic you want to visualize. Hence, they indicate what you actually see. Examples are: points, boxplots, lines, polygons etc.
- **stats** (statistical transformations) are the statistics you want to calculate for the data. They allow us to summarize data in many useful ways. A scatter plot for example simply uses the identity, a bar plot uses frequencies, etc.
- **scales** map values in data space to values in aesthetic space. They define colour, size or shape and are used to create legends and axes.
- **coord** (coordinate system) indicates how data coordinates are mapped to the plane of graphic. Examples are axes and gridlines. Usually the Cartesian coordinate system is used.
- **facet** (faceting specification) breaks up the data into subsets and reproduces the graphic on each subset.
- **theme** defines the fonts of all labels, the shape of ticks and the background.

The grammar of graphics is the basis of a well-structured and logical graphic language. There are, however, tasks that the grammar of graphics does not do for you. It does not:

- tell you which graphic is appropriate for your data and question,
- specify what exactly a graphic should look like (background color, font size, etc.),
- allow interaction, movement, 3D. This can be achieved using the `ggvis` package (see Section 6.2.11).

The function `ggplot()` needs some time to get used to, especially when coming from the `graphics` package. To make `ggplot2` a bit easier for beginners, there are the `qplot()` functions. `qplot()` stands for “quick plot” and tries to combine the grammar of graphics with the syntax from the `plot` function of the `graphics` package. When using `qplot()`, it is however not possible to use the whole range of functionalities provided by the grammar of graphics. That’s why we would recommend you to work with `ggplot()` right from the start. Additionally, the `ggplot2` naming convention for the layers simplifies the usage of the language: All geoms have names of the form `geom_<geom-name>()`, all statistics are called `stat_<stat-name>()`, and so on.

The layers are concatenated with the “+” operator. This makes it easy to interactively update a plot. You might for example first use the `geom_point()` function in order to construct a scatter plot between two variables. Then, since you suspect a positive relationship between the two, you add a `geom_smooth()` in order to draw a line through the point cloud that visualizes this relationship (see Section 6.2.8). Basically, there are always two ways to construct a graph: Either by specifying a `geom` layer or by specifying a `stat` layer. See the line plot (Section 6.2.3) for an example.

Now we will introduce different types of plots to you. Through the whole section we are going to work with the Google Analytics dataset. So let’s get started!

### 6.2.2 Column or bar plot

Imagine that we are interested in the distribution of gender in the Google Analytics data. We can use a column or bar plot to visualize the distribution of variables with nominal or ordinal scale of measurement. In order to avoid misleading interpretations, it is necessary to include the zero in the y-axis. Additionally, there should be no overlapping columns and the number of columns should not be too large.

```r
library(ggplot2)
ggplot(analytics) +
  geom_bar(aes(x = userGender, fill = userGender), color = NA) +
  xlab("Gender") +
  ylab("frequency") +
  ggtitle("Bar plot") +
  scale_fill_manual(values = c("female" = "#2B4894", "male" = "#DD9123"),
                    guide = FALSE)
```

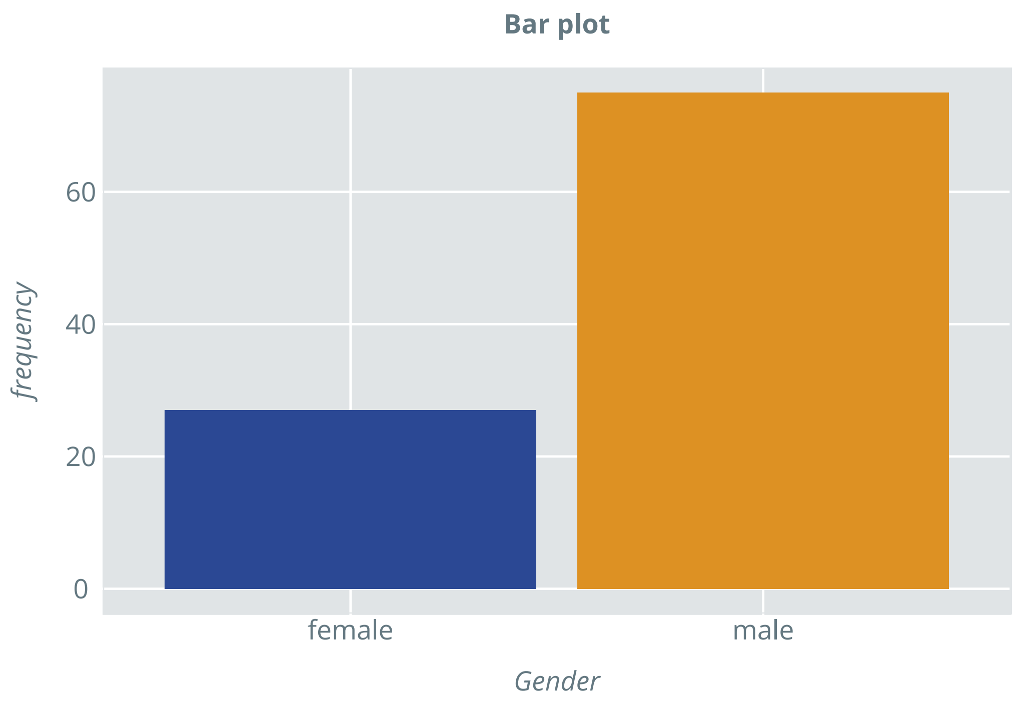


**Figure 8** – Gender distribution in the Google Analytics dataset

Note that the colors have been chosen such that their brightness differs considerably. As a result, the difference between the colors is visible for colourblind people or if printed in grayscale.

By the way: It makes a difference whether you specify `color`, `fill` etc. inside or outside `aes()`. If you define it inside (like `fill` in our example bar plot), you can choose a variable from the data to define groups which should be colored differently – for example `userGender`. But sometimes you may want to change the color of the whole plot. Then you specify it outside `aes()`. This applies to all types of plots. See how the bar plot changes when we play around a bit:

```
ggplot(analytics) +
  geom_bar(aes(x = userGender), fill = "yellow", color = "darkgreen") +
  xlab("Gender") +
  ylab("frequency") +
  ggtitle("Bar plot (color and fill defined outside aes)")
```

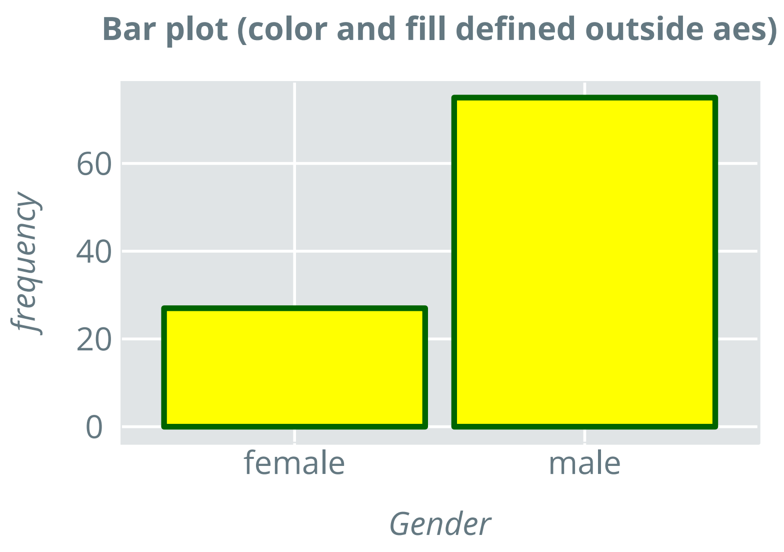


**Figure 9** – Bar plot where `color` and `fill` are specified outside `aes`

```
ggplot(analytics) +
  geom_bar(aes(x = userGender, fill = userGender), color = "black") +
  xlab("Gender") +
  ylab("frequency") +
  ggtitle("Bar plot (color defined outside aes)") +
  scale_fill_manual(values = c("female" = "#2B4894", "male" = "#DD9123"),
                    guide = FALSE)
```

### 6.2.3 Line plot

Suppose we are suspecting a relationship between the duration of a session and the number of pageviews. An easy way to examine this relationship is by plotting a *line plot* with the independent variable (duration of the session) on the x-axis and the dependent variable (number of pageviews) on the y-axis.

```
ggplot(analytics, aes(x = sessionDuration, y = pageviews)) +
  geom_line() +
  geom_rug(sides = "b") +
  xlab("Duration of Session") +
```

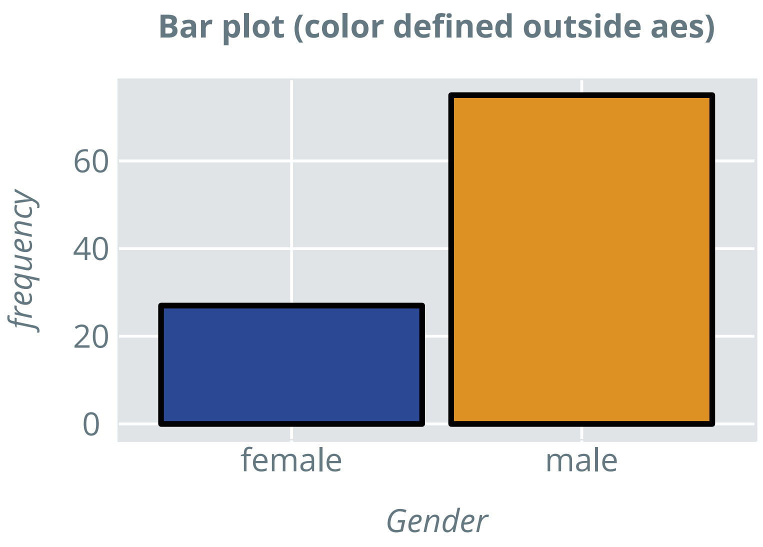


**Figure 10** – Bar plot where `color` is specified outside and `fill` is specified inside `aes`

```
  ylab("Number of Pageviews") +
  ggtitle("Line plot")
```

The same plot would be created by the following code:

```
ggplot(analytics, aes(x = sessionDuration, y = pageviews)) +
  stat_identity(geom = "line") +
  stat_identity(geom = "rug", sides = "b") +
  xlab("Duration of Session") +
  ylab("Number of Pageviews") +
  ggtitle("Line plot")
```

As expected, the amount of pageviews increases as the duration of session gets longer. There is, however, quite some variation in the data, leading to spikes and dips in the line. For longer durations of sessions it seems like the relationship loses its wiggliness. However, this is an artefact caused by only a small amount of observations with long session durations. In order to visualize this effect, small ticks which indicate the observations are added to the x-axis by using the function `geom_rug()`[25].

Figure 12 shows a somewhat different linegraph of a survival function estimated by the Kaplan-Meier estimator[26]. It will be used in order to demonstrate some pitfalls of drawing plots. Originally, a survival function was used to display the probability of survival over time or, framed differently, how rapidly people died in a certain sample. Transferred to a somewhat nicer context, you can imagine a pool of customers. The survival function now tells you how fast those customers convert, i.e., for example buy a certain product. At the

[25]Be careful if you are handling big data. You shouldn't use a rug plot since it needs quite some time to load when included in a presentation.
[26]The R Code for the graphic is not provided. When working with survival data, the package `survival` is needed.

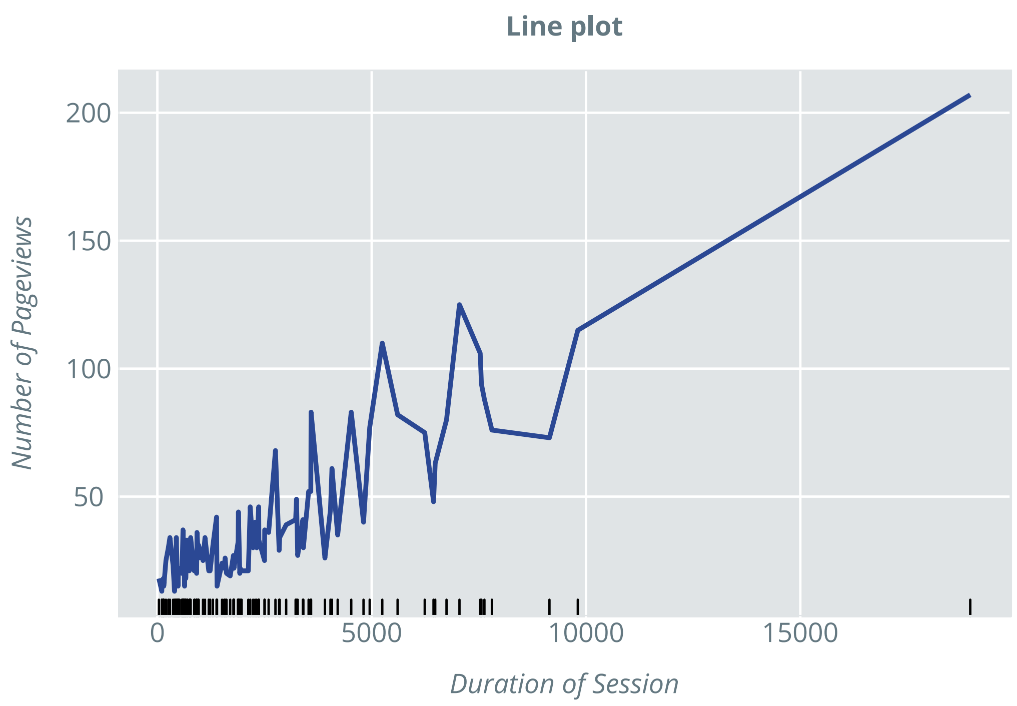


**Figure 11** – Line plot with ticks for each observation

beginning, a percentage of 100% of the customers has not converted yet. This percentage declines as time goes on, meaning that one customer after the other converts.

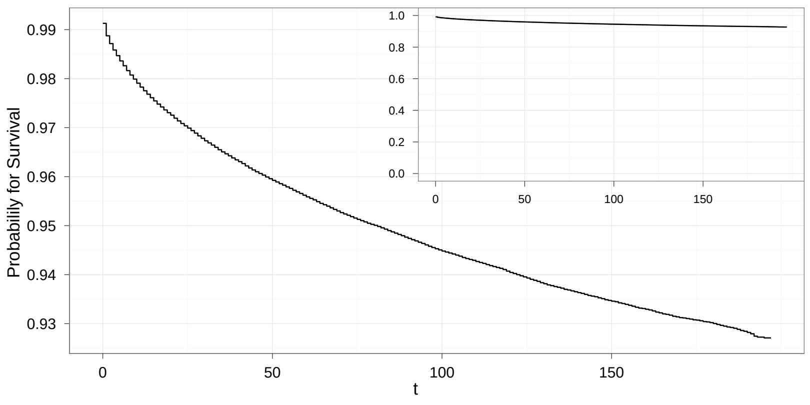


**Figure 12** – Survival function with Kaplan-Meier estimator

Now, the two plots in Figure 12 show the same facts using a different scaling of the ordinate. Accordingly, interpretations made on the fly differ enormously: The bigger plot suggests a high conversion rate whereas the smaller plot in the upright corner gives the impression of a very small conversion rate. What is right? If possible, you should always include the zero in order to give the reader the chance to judge him- or herself.

Another aspect of line graphs is the number of lines. Sometimes it is very tempting to include all the measures you want to compare in one graph. This, however, can lead to ambiguous interpretations, especially when in a print version colors are turned into grey scales and lines cannot be distinguished anymore.

In RStudio, graphics are displayed in the "Plots" pane and can be browsed simply by hitting the blue arrows. In R itself, they are shown in a separate window ("R Graphics"). There, each new graph overrides the previous one. In both cases, plots can be saved in several graphical formats. Since the plots produced by R are vector graphics they should – if possible – be saved and processed in vector formats (e.g. Metafile, postscript or pdf). The advantage of vector graphics is that they can be deliberately resized without loss of quality.

### 6.2.4 Pie chart

Alternatively, for the same kind of data, a *pie chart* can be used. They just look nice - nothing more. But also not less, and you see at once that something complete is divided into parts. The objection that the human eye is better able to compare lengths than angles is absolutely clear. That is why some people advice against the usage of pie charts at all, e.g., the developers of ggplot2. Therefore, this package does not offer a visually appealing implementation of pie charts.

If you still want to plot pie charts, since you want to communicate the message that all indistinguishable pie pieces are similar and differences are irrelevant, then the package plotly provides a pretty alternative.

The following code draws a pie chart for the distribution of gender in the Google Analytics data:

```
library(plotly)
analytics %>%
  group_by(userGender) %>%
  summarise(count = n()) %>%
  plot_ly() %>%
  add_pie(labels = ~userGender,
          values = ~count,
          marker = list(colors = c("#2B4894", "#DD9123")),
          textposition = "outside",
          textinfo = "label+percent",
          direction = "clockwise")
```

The syntax of plotly differs slightly from the syntax of ggplot2. In order to call a variable out of a dataset, use the tilde as prefix (`~variable`). The colors of the segments are set by the optional parameter `marker` which has to contain as many colors as there are `labels`. In R, colors can either be chosen from one of the many prespecified colors (`colors()` lists them all) or be specified by hexadecimal code preceeded by a hash.

### 6.2.5 Histogram

Now we are interested in the distribution of the average time on page. So we are looking at a continuous measurement and there is a sufficient number of observations available. In this case we can use the *histogram*:

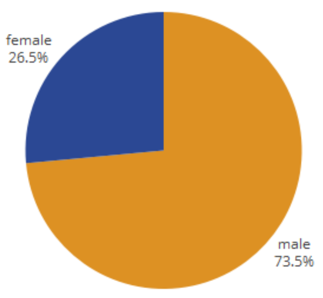


**Figure 13** – Piechart for gender distribution

```r
ggplot(analytics) +
  geom_histogram(aes(x = avgTimeOnPage, y = ..density..),
                 binwidth = 20,
                 fill = "grey",
                 color = "#2B4894") +
  xlab("Average Time on Page") +
  ggtitle("Histogram")
```

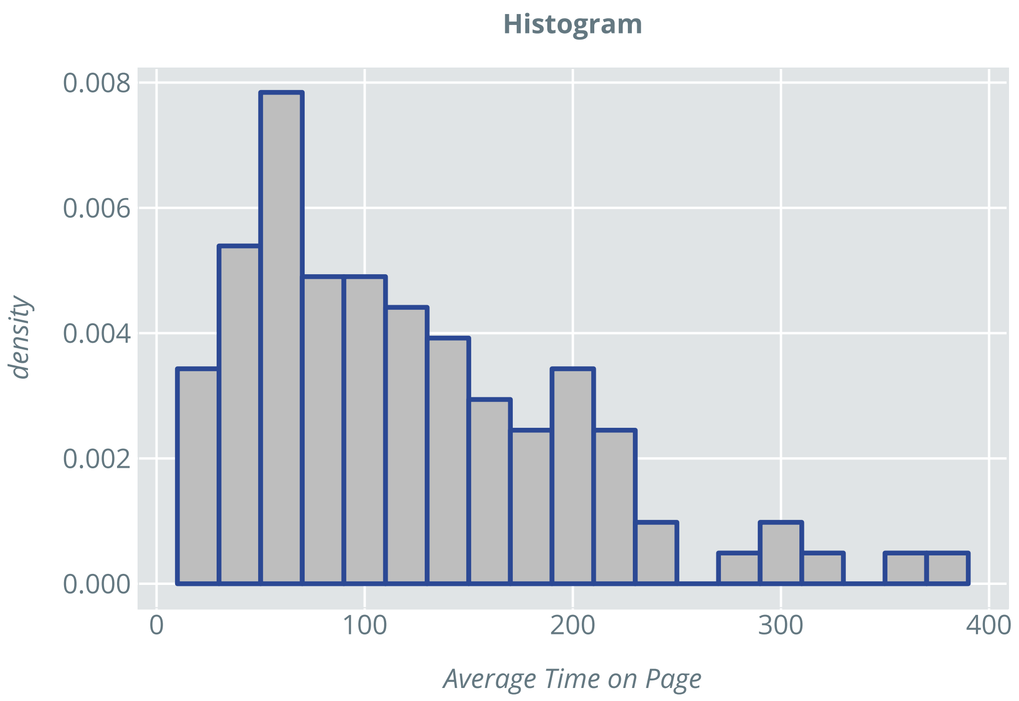


**Figure 14** – Density of the average time on page from the Google Analytics Dataset

By setting the optional aesthetic `y = ..density..` the relative frequencies instead of the absolute frequencies are printed. Additionally, the width of classes is set with the optional parameter `binwidth`, and the parameter `colour` specifies the color of the borders of the bars. You can go ahead and see what happens if you change these parameters yourself!

### 6.2.6 Box plot

Concerning the distribution of the average time on page, it could be interesting to see if there are differences between women and men. This is a question that can be answered by looking at the corresponding *box plots*[27]. In `R`, box plots are generated by using the function `geom_boxplot()`. As aesthetics, parameter `y` indicates the continuous variable whose values are separated and plotted according to each factor level of the variable specified by the parameter `x`.

By default boxplots are printed vertically. This can be changed by adding the function `+ coord_flip()` which exchanges the x- and y-axis. If the data contain an extreme value and this value is included in the plot, it becomes illegible. With `+ coord_cartesian(ylim = c(a, b))`, you can restrict the y-axis to the range from a to b and enhance legibility.[28]

```
ggplot(analytics, aes(x = userGender, y = avgTimeOnPage)) +
  geom_boxplot(fill = "#6b99ce") +
  xlab("Gender") +
  ylab("Average Time on Page") +
  ggtitle("Boxplot")
```

Note that using a boxplot only makes sense with an adequate number of observations. If the amount of observations is rather low, a scatter plot can be more informative than a boxplot. Also the number of observations should always be reported in the text.

### 6.2.7 QQ plot

In inferential statistics there is an important requirement for many tests: the normal distribution of the data. Especially for small datasets, tests of this assumption are not applicable and for big datasets their validity can be questioned as well. Alternatively, it is common to use *QQ-Plots*[29] in order to visually test the assumption of normality. The following figure shows the QQ plot for the average time on page from the Google Analytics dataset.

```
ggplot(analytics, aes(sample = avgTimeOnPage)) +
  ggtitle("QQ Plot") +
  stat_qq()
```

[27]Of course, box plots can also be used to look at a single variable's distribution independent of any groups.

[28]Another way to do this is `+ ylim(a, b)` resp. `+ xlim(a, b)`. But be careful: `coord_cartesian` just cuts the plot, whereas `xlim` and `ylim` cut the data before the plot is created. If any computed value appears in the plot (e.g., the box in a box plot), `xlim` and `ylim` will lead to a bias because not alle the data are included!

[29]Quantile-Quantile plot, where the quantiles of two variables are plotted against each other in order to compare their distributions.

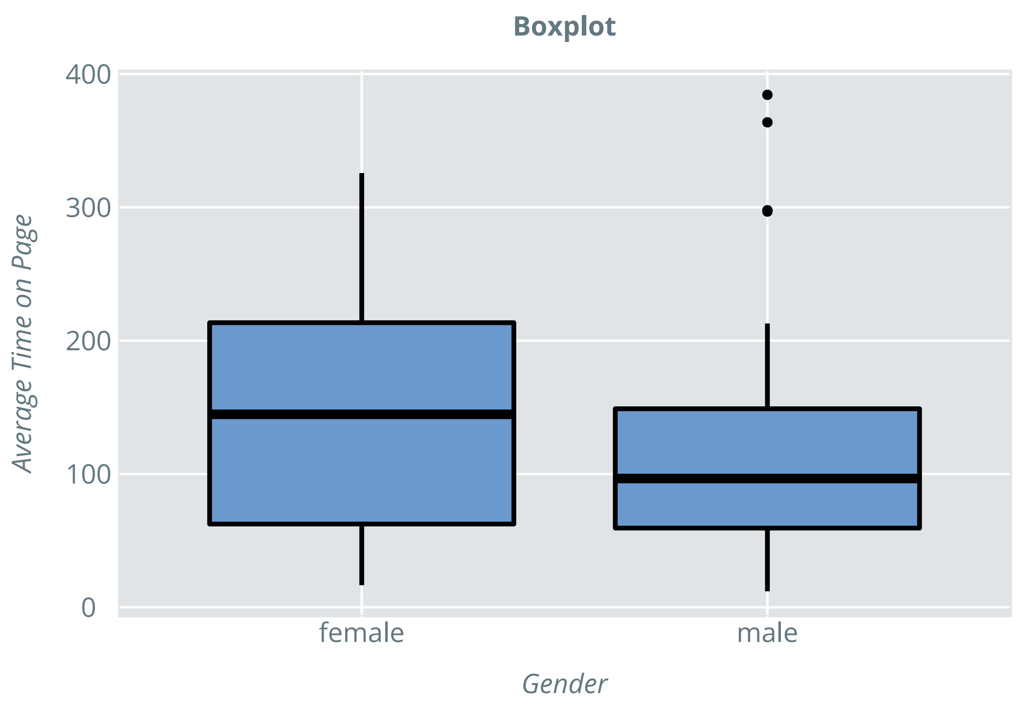


**Figure 15** – Distribution of the average time on page by gender from the Google Analytics Dataset

The function `stat_qq()` draws the empirical quantiles of the variable (ordinate) versus the theoretical quantiles of the standard normal distribution (abscissa). Perfect normality would be represented by a straight line ranging from the lower left corner to the upper right corner of the plot (best-fit line). Deviations of the points from the line mark a deviation from normality. So the QQ plot for the average time on page shows us that the data is not perfectly normally distributed. It is, yet, not heavily skewed either. Note that in this case, the functional scope of the `ggplot` package and the `graphics` package differ considerably. In `graphics`, it is possible to add the best-fit line. This, however, is not straightforward in `ggplot` since it contradicts the logic of the grammar of graphics.

### 6.2.8 Scatter plot

Statistics is about discovering regularities in data in order to extract and compress information. So let's check whether there is a relationship between the duration of session and the number of pageviews. We could suspect a positive relationship, since the longer a session, the more time a person has to check out several pages. Let's examine the relationship between these two metric variables graphically by using a *scatter plot* complemented by predictions made from a local regression. Therefore, the `geom_point()` function together with the `geom_smooth()` function is used.

```
ggplot(analytics, aes(x = sessionDuration, y = pageviews)) +
  geom_point() +
```

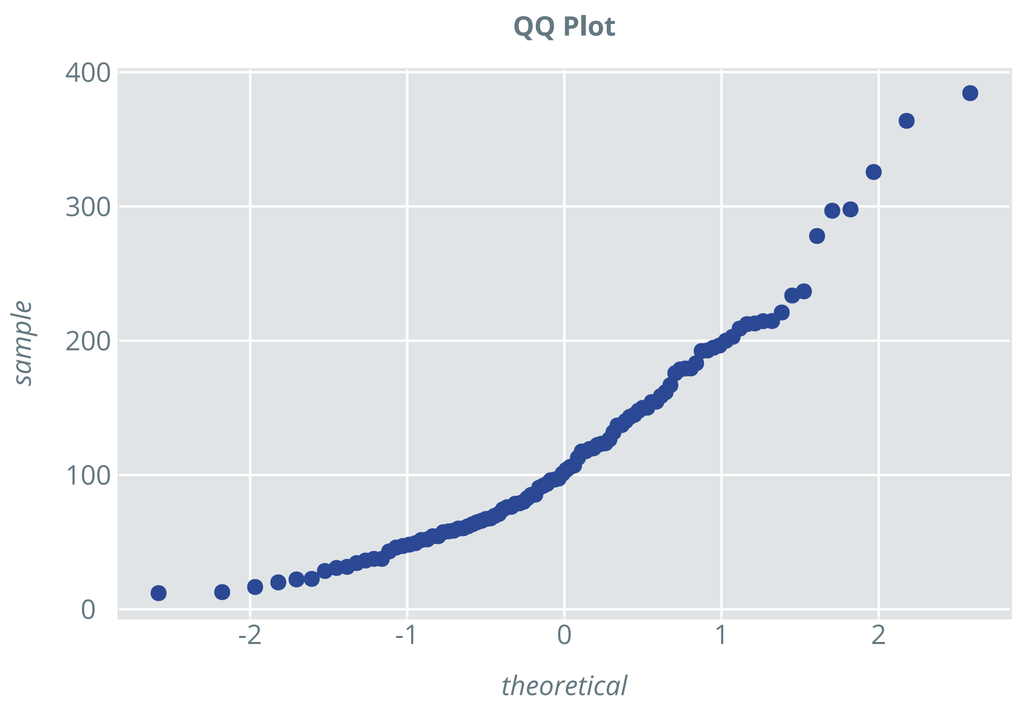


**Figure 16** – Test of normality assumption via QQ plot for `avgTimeOnPage` from the Google Analytics dataset

```
  geom_smooth() +
  xlab("Duration of Session") +
  ylab("Number of Pageviews") +
  ggtitle("Scatter plot")
```

`R` by default draws filled circles which can be changed by using the optional parameters `shape` and `size` in `geom_point()`.

### 6.2.9 Facets

Sometimes you may want to create the same plot for several groups and arrange them row- or columnwise. You can achieve this by adding a `facet_grid` to a plot:

```
# Plot without facets
p <- ggplot(analytics) +
  geom_histogram(aes(x = sessionDuration), color = NA)
p
```

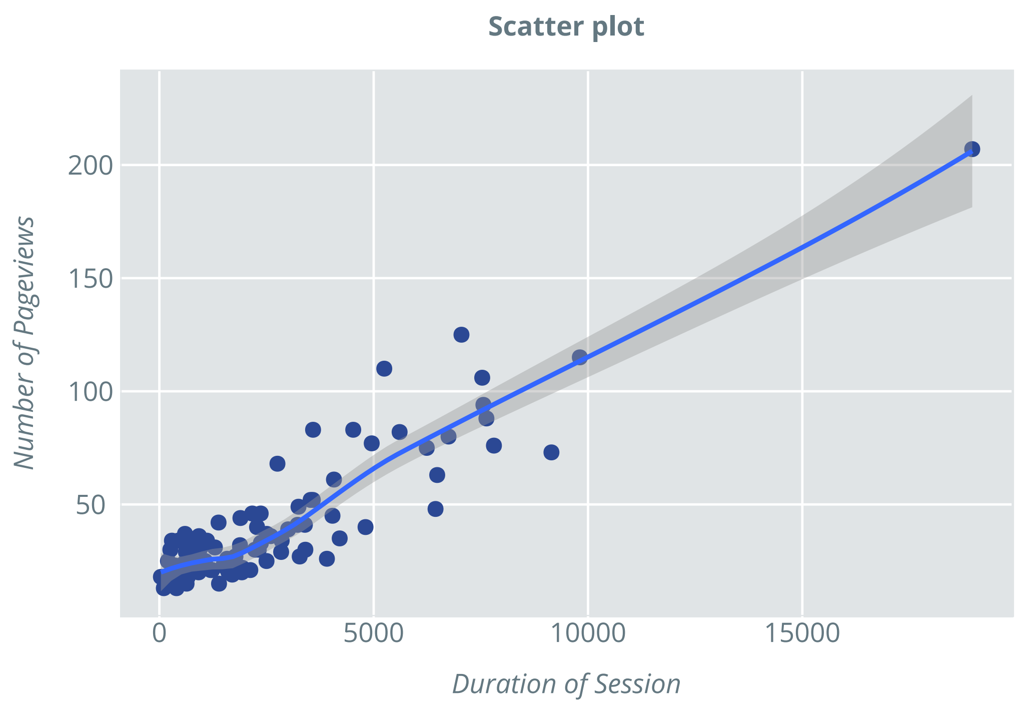
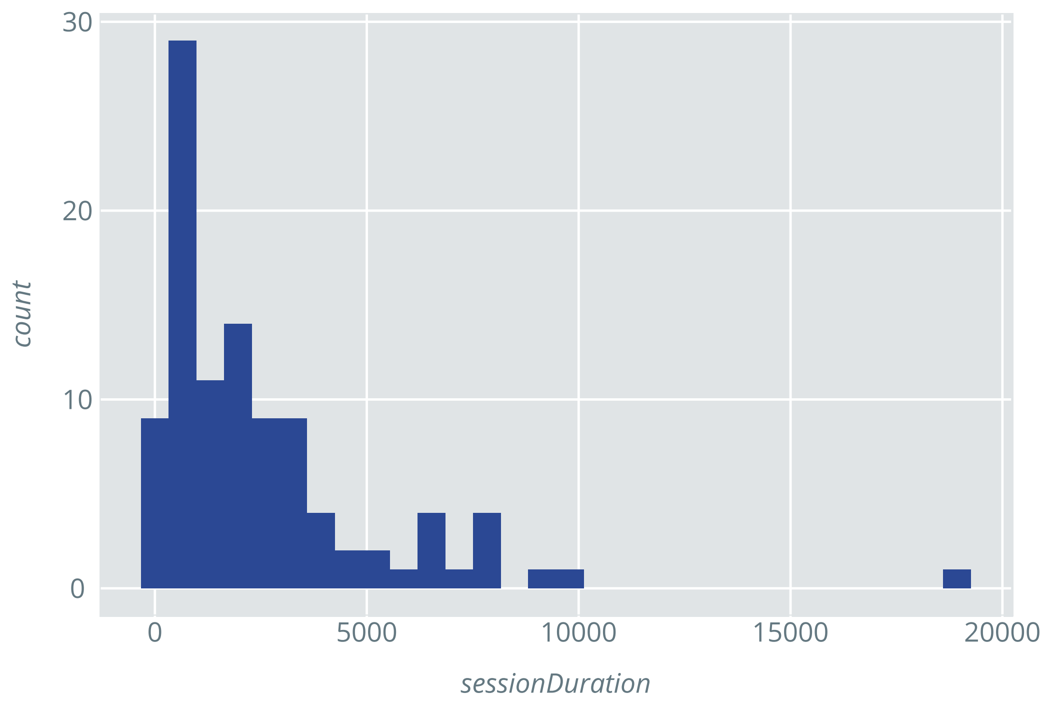
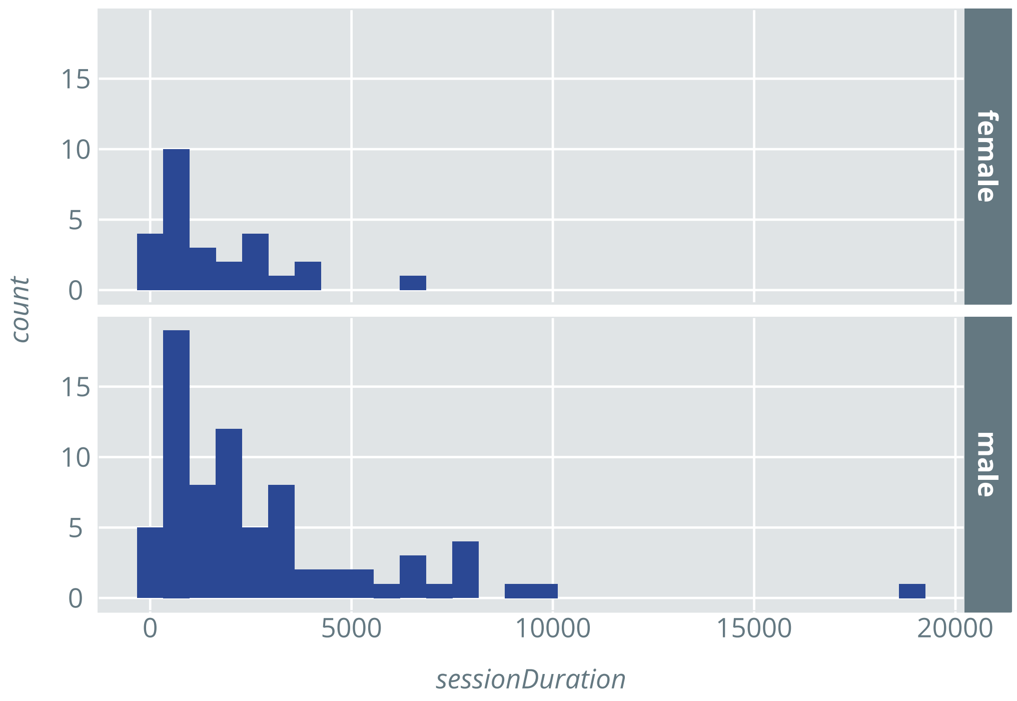


**Figure 17** – Scatter plot visualizing the association between the session duration and the number of pageviews

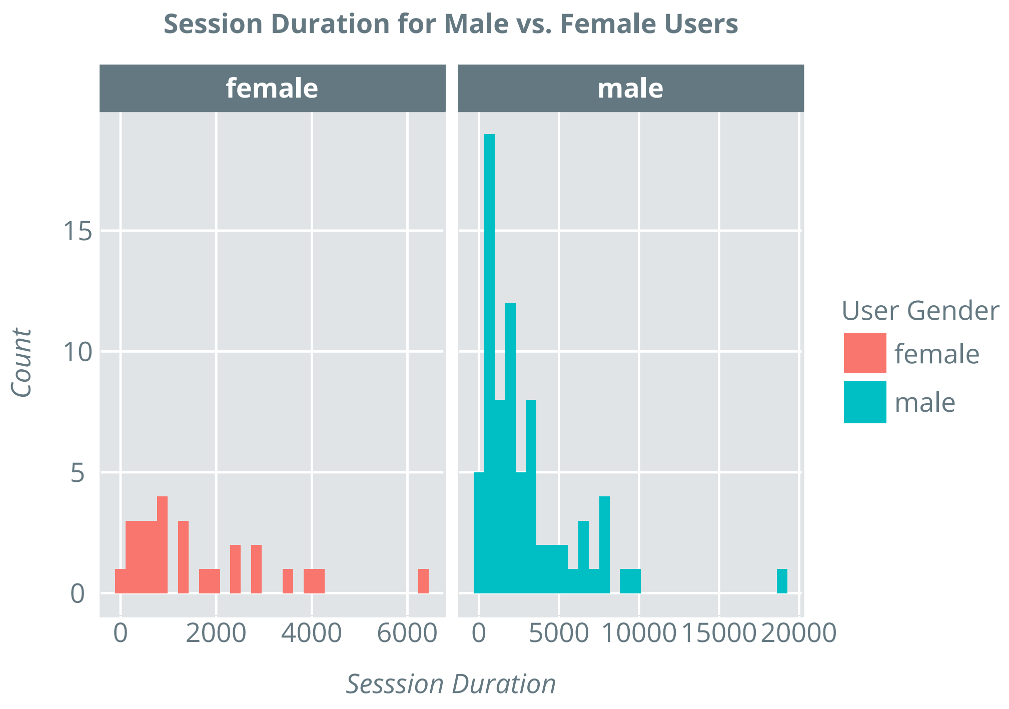


```r
# Same plot with facets
p + facet_grid(userGender ~ .)
```



You can also arrange the plots columnwise, add custom fill colors and labels, allow the axes to vary between the plots and much more.

```r
ggplot(analytics) +
  geom_histogram(aes(x = sessionDuration,
                     fill = userGender), # Different colors by gender
                 colour = NA) +
  facet_grid(. ~ userGender, # Show different genders in columns
             scales = "free_x") + # Allow x-axis to vary
  labs(x = "Sesssion Duration",
       y = "Count",
       title = "Session Duration for Male vs. Female Users",
       fill = "User Gender")
```



For more information see the the help page via `?facet_grid`.

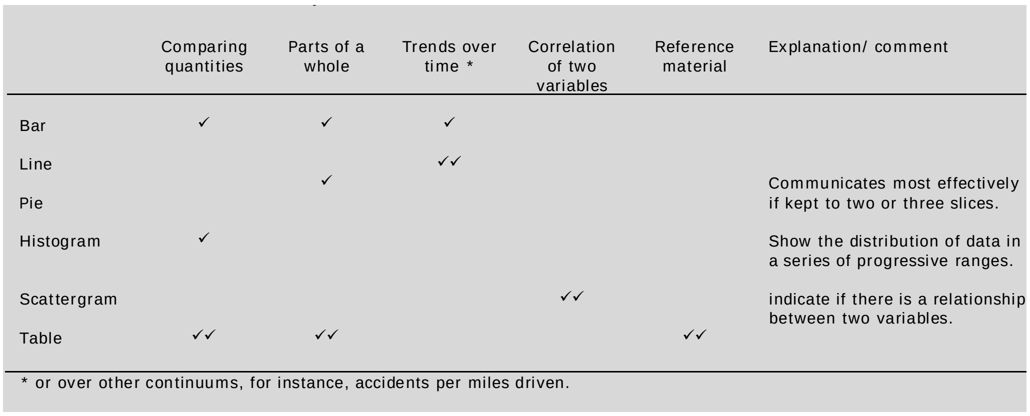
### 6.2.10 Additional information on `ggplot2`

All plots presented above can be further adjusted. Type `help("...")` in order to obtain additional information about the functions. Moreover, a multitude of adjustments for the graph window can be made (e.g. enlargement of the picture margins, splitting into several plotting regions, etc.). You can get an overview with `help(theme)`.

Figure 18 gives an overview about which kind of graph to use for which kind of information[30].

In order to save a graph, the functions `bmp()`, `jpeg()`, `png()`, `tiff()`, `pdf()`, `postscript()` or `win.metafile()` can be combined with `dev.off()`.

[30]Reference: Bigwood S., Spore M.: "When to use numeric tables and why - Guidelines for the brave" http://www.plainfigures.com/downloads/when_to_use_tables_and_why.pdf accessed 29.05.2017.



| | Comparing quantities | Parts of a whole | Trends over time * | Correlation of two variables | Reference material | Explanation/ comment |
|---|---|---|---|---|---|---|
| Bar | ✓ | ✓ | ✓ | | | |
| Line | | | ✓✓ | | | |
| Pie | | ✓ | | | | Communicates most effectively if kept to two or three slices. |
| Histogram | ✓ | | | | | Show the distribution of data in a series of progressive ranges. |
| Scattergram | | | | ✓✓ | | indicate if there is a relationship between two variables. |
| Table | ✓✓ | ✓✓ | | | ✓✓ | |

* or over other continuums, for instance, accidents per miles driven.

**Figure 18** – Overview on graph types and their features

### 6.2.11 Interactive graphics with `ggvis`

When dealing with static graphics, `ggplot2` is very mighty and in our opinion the best choice. There is, however, one feature it does not provide: interaction. That's where `ggvis` comes into play. It allows us to build interactive graphics relatively simple. The following is only intended as basic introduction since `ggvis` is very comprehensive if you go into detail. The help files and examples are yet easy to understand and several introductions exist on the internet[31]. Note that, when working with `ggvis`, each interactive plot must be connected to a running `R` session. Otherwise, the interactive graphics are not interactive anymore. We will present some examples later on but here, too, printing on paper does unfortunately not allow interaction.

In a sense `ggvis` is similar to `ggplot2` since graphics are constructed layer by layer. Layers are concatenated by using the pipe operator `%>%`. There exist five basic layers in `ggvis`:

- `layer_points()` drawing points in a plot like e.g. in a scatter plot,
- `layer_paths()` connecting observations by lines,
- `layer_ribbons()` printing filled areas,
- `layer_rects()` drawing rectangles,and
- `layer_text()` printing text to where the marks are.

Furthermore, there are four so-called compound layers that combine two or more operations in one call:

- `layer_lines()` first orders the observations according to their x values by using `arrange()` and then prints the observations using `layer_paths()`.
- `layer_histograms()` and `layer_freqpolys()` are equivalent to binning the observations first and then printing rectangles or lines according to the count of observations in each bin.
- `layer_smooths()` is equivalent to calculating predictions of a smoothed model first and then printing them with `layer_paths()`.

[31] Here you can find further links to various topics around `ggvis`: http://ggvis.rstudio.com/. The following section is mainly based on the `ggvis-basics` vignette (http://ggvis.rstudio.com/ggvis-basics.html).

It is of course possible to lay multiple layers on top of each other, just like we are used to from `ggplot2`.

There is two pecularities about `ggvis`. First, if you want to call on a variable out of a dataset, use the tilde as prefix (`~variable`). And second, if for example the color or size of some marks is fixed and not to be set according to the levels of a factor or the like, := is used instead of =.

Well, the big advantage of `ggvis` is the interaction it adds to graphics. Therefore it provides several input elements:

- `input_slider()` to enter a value by moving a slider,
- `input_checkbox()` to set an argmuent `TRUE` or `FALSE` by clicking a checkbox,
- `input_checkboxgroup()` to control several parameters through a group of checkboxes,
- `input_numeric()` to set a numeric argument through a spin box,
- `input_radiobuttons()` makes it possible to choose one from a set of options,
- `input_select()` creates a drop-down text box to chose from and
- `input_text()` enables you to input text.
- Additionally, there are the functions `left_right()` and `up_down()` which allow you to use keyboard controls in order to set numerical arguments.

Now let's see some examples. In the following we will redo some graphics from above, introducing interaction by using `ggvis`.

So first we want to construct a line plot where we can name the marks flagging the observations. Additionally, we add a slider in order to set the opacity of the marks and line (see Figure 19). We can achieve this by using the `input_text()` function for the naming of the marks. It takes as arguments a predefined text, in this case "dot", and a label. In order to regulate the opacity, we work with the `input_slider()` function. Here we need to specify the minimum and the maximum value of the slider. Additionally, we can set a default value and a label. Unfortunately, it does not seem to be possible to add rugs to the plot that show the distribution of the observations.

```r
# Line plot with customized marks
library(ggvis)
inpText <- input_text("dot", label = "Name the Marks")
analytics %>%
  ggvis(~sessionDuration, ~pageviews,
        opacity := input_slider(0, 1, value = 0.7, label = "Set the opacity"))
            %>%
  layer_lines() %>%
  layer_text(text := inpText) %>%
  add_axis("x", title = "Duration of Session") %>%
  add_axis("y", title = "Number of Pageviews")
```

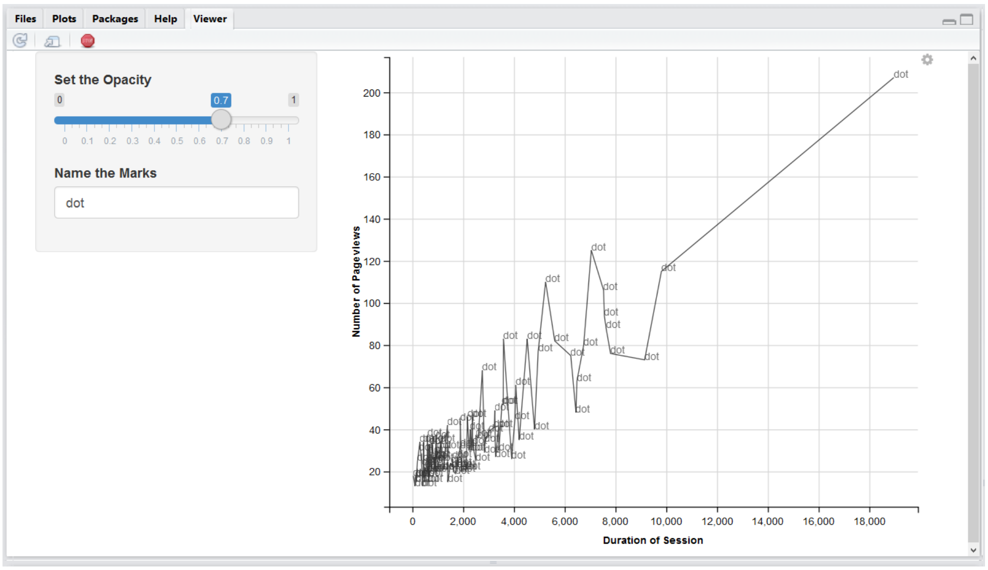


**Figure 19** – Line plot with possibility to name the marks and set the opacity of the symbols and line

Now we want to construct a histogram, where it is possible to adjust the binwidth ourselves (see Figure 20). We can do this again with the `input_slider()` command. We added the additional `step` argument. Here it seems unfortunately not to be possible to print the density instead of the counts.

```
# Histogram with binwidth as slider input
inpBinwidth <- input_slider(5, 200, value = 100,
                            step = 10, label = "Adjust Binwidth here")
analytics %>%
  ggvis(~avgTimeOnPage, fill := "lightgrey") %>%
  layer_histograms(width = inpBinwidth) %>%
  add_axis("x", title = "Average Time on Page") %>%
  add_axis("y", title = "Count")
```

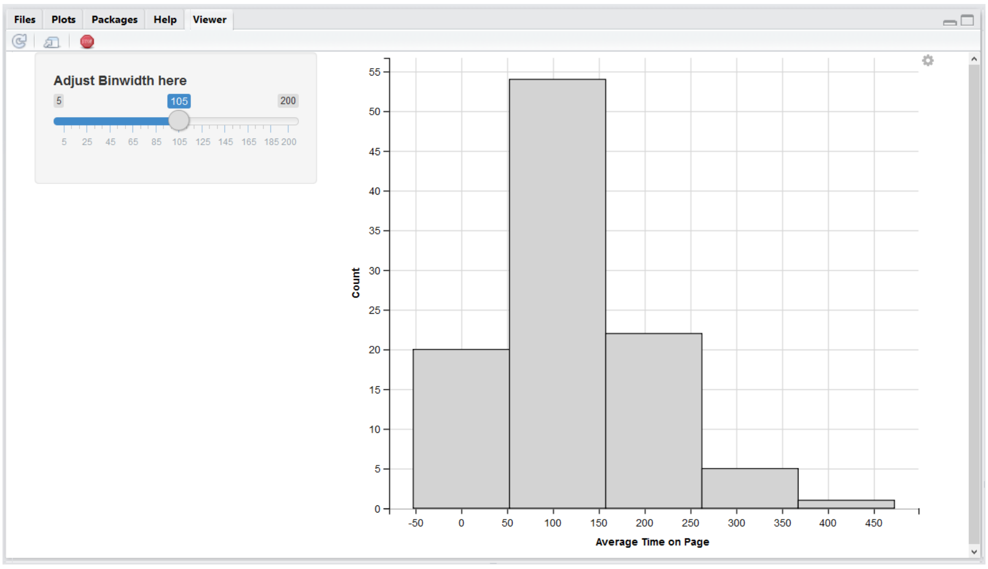


**Figure 20** – Histogram with possibility to adjust binwidth

Last but not least, we want to construct a scatter plot including a regression line where it is possible to choose the color of the points and line through a drop-down menu. Additionally, we construct a checkbox that lets us choose whether we want to make predictions based on a linear regression (default) or on a local regression (see Figure 21). Note the `map` argument of the `input_checkbox()` function that converts the input values `TRUE` or `FALSE` to "loess" or "lm", arguments which the `model` argument of the `layer_model_predictions()` function understands.

```
# Scatter plot with customized colors and smoothed line built by either linear
# or local regression
inpColor <- input_select(choices = c("Blue" = "#2B4894",
                                     "Red" = "#bc423a",
                                     "Green" = "#4B9081"),
                         label = "Color")
inpModel <- input_checkbox(label = "Local Regression",
                           map = function(val) if (val) "loess" else "lm")
analytics %>%
  ggvis(~sessionDuration, ~pageviews, fill := inpColor) %>%
  layer_points() %>%
  layer_model_predictions(stroke:= inpColor,
                          fill := inpColor,
                          model = inpModel) %>%
  add_axis("x", title = "Duration of session") %>%
  add_axis("y", title = "Number of pageviews")
```

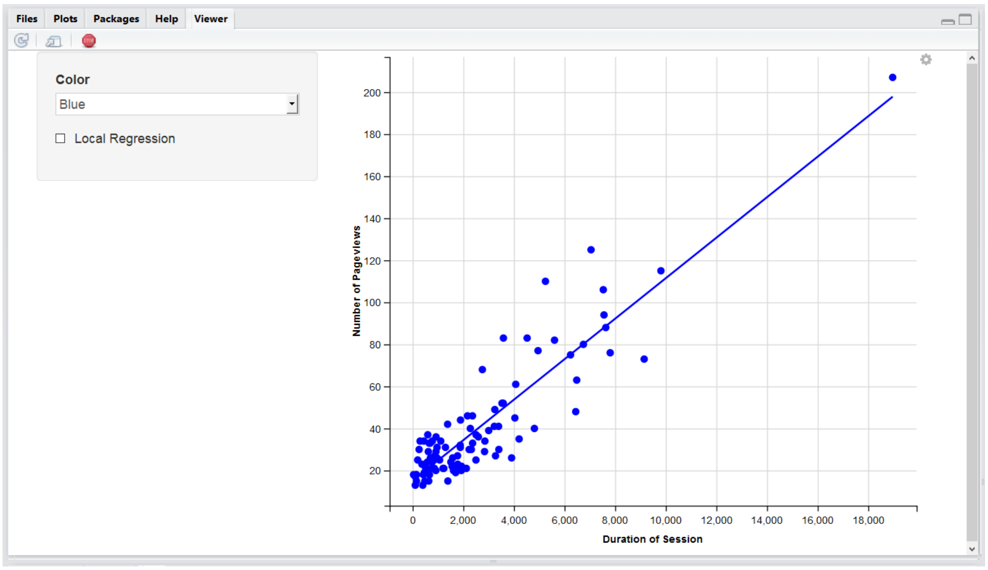


**Figure 21** – Scatter plot with possibility to choose color and type of line

There are many more functions to create interaction. It is for example possible to create plots where the x-coordinate is shown once you hover with your mouse over an observation. Or it is possible to adjust the size of the dots in a scatter plot by using the arrow key to the left or right. The best to explore the whole functionality of `ggvis` is to look at the help pages and try to rebuild the examples with your own data. So, have fun!

### 6.2.12 Graphics: lessons learned

So here are some points to wrap it up.

- Even if it may be tempting, try not to use **3D plots**. Instead of making things clearer they are more likely to confuse. Using 3D elements makes the plot more complex without adding any information. Confusion can also be caused by excessive use of auxiliary lines. There are, however, cases where they are quite useful.
- If you are thinking about chosing fancy **colors** for your plots, keep in mind that in print they might turn out in black and white. So one criterion of choice should be that they differ considerably in their grey scales. Also there are quite a few people out there who are color-blind. So sometimes it makes sense to distinguish lines or bars by type and texture.
- **Scales** should generally include the zero as reference point. Additionally, a wiggle of the axis to indicate a jump in measurement is confusing and should be avoided. For enhanced legibility, numbers at the axes should be rounded. If the numbers are very big, it can be useful to show them “in thousand” or “in millions” instead of writing these large numbers to the x-axis.
- We have also seen that the **number of observations** underlying a certain part of a plot can possibly explain some odd behavior of e.g. a line plot. In this case, adding tick marks for each observation can be helpful. For boxplots as well it is useful to report the number of observations, since they contain a collection of summary statistics.

- **Labelling and captions** should be kept to a minimum and at the same time be explicit and unambiguous.
- `R` produces high quality **vector graphics** so they should be saved and processed in vector formats (e.g. Metafile, postscript or pdf).
- Coming to interaction, with `ggvis` there exists a mighty but easy to use tool to add interactive elements to your plot. However, note that each interactive plot must be connected to a running `R` session. So here, similar to 3D and other fancy graphical elements, you should always question whether interaction is really what you need.

`ggplot2` and its layered grammar of graphics is a great package to build static graphics. Even though it is hard to understand in the beginning, once you have got it, you can build nice and comprehensive graphics. Since quite some functionalities are not available in `ggvis`, `ggplot2` is to be preferred. It is, however, not possible to introduce interaction into a `ggplot2` plot.

In general: Whatever you plot - question what you do!

## 6.3 Presenting numbers in tables

### 6.3.1 How to express numbers in your writing?

One thing about numbers is whether or not to spell them out when you use them in your writing. There is one simple and agreed-upon rule: all numbers smaller than 10 are spelled out. All other numbers are written as digits. Coming to percentages, it is up to you to choose whether or not you want to write “one percentage” or “1%”. In a way, the spelled-out way looks a little more formal. Whether to round “big” numbers or not, is a question about the necessity of the exact information. Sure is that “10 thousand people” is easier to read and better to grasp than “10,324 people”. Especially when you want to talk about sums and differences, the sum of 12,6 million plus 11,9 million is calculated more easily than the sum of 12,573,981 and 11,894,397. Two important questions to answer when thinking about displaying numbers are: “Does substantial information get lost by the way of presentation?”. And: “How can the way of presentation help to reinforce the point I want to make?”.

### 6.3.2 Format of tables

Table 1 shows two tables about the population’s development in a country over the years. In the left table, detailed information (exact number of residents) is provided every 10 years from 2000 to 2090. In the right table, information is shown only for the years 2000, 2020, 2050 and 2090. Additionally, numbers are rounded. Even though the right table holds less information, interpretation is nevertheless easier: The population increases from 2000 to 2020, then stagnates till 2050 and rises again till 2090.

| Year | Population |
|---|---|
| 2000 | 2,013,245 |
| 2010 | 2,342,231 |
| 2020 | 2,656,621 |
| 2030 | 2,593,982 |
| 2040 | 2,599,654 |
| 2050 | 2,644,211 |
| 2060 | 2,731,456 |
| 2070 | 2,792,935 |
| 2080 | 2,896,474 |
| 2090 | 2,903,085 |

| Year | Million |
|---|---|
| 2000 | 2.0 |
| 2020 | 2.7 |
| 2050 | 2.6 |
| 2090 | 2.9 |

**Table 1** – Population of a country over time

Still, if interest lies in the yearly population, Figure 22 gives deeper insights in the development of residents.

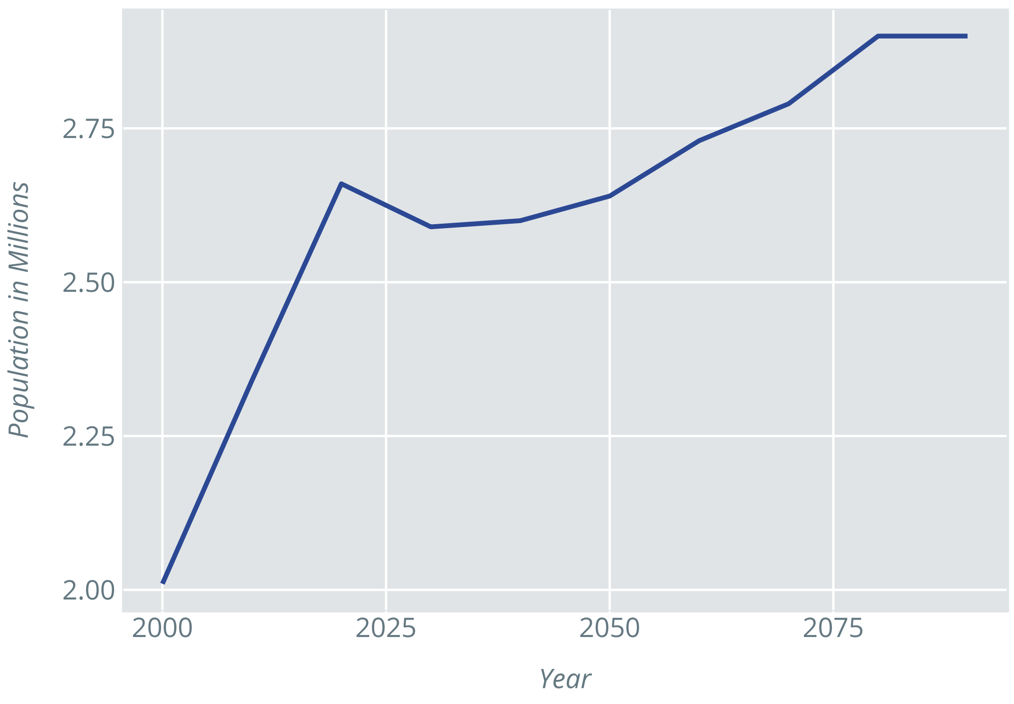


**Figure 22** – Development of a population over the years

Coming to the alignment of row and column names as well as the actual numbers, the rownames should be left-aligned and the column names and numbers right-aligned. The two tables in Table 2 illustrate the effect of this simple rule: the right hand table is more readable than the left hand table.

| | Mean |
|---|---|
| Group 1 | 13,0 |
| Group 2 | 115 |
| Group 3 | 2,67 |
| Group 4 | 10,1 |

| | Mean |
|:---|---:|
| Group 1 | 13,0 |
| Group 2 | 115,0 |
| Group 3 | 2,7 |
| Group 4 | 10,1 |

**Table 2** – Means of four groups

Row sums should be printed in the right most column, column sums should be printed in the last row (see for example the frequency table – Table 3 – with the column sums in the last row).

If e.g. certain characteristics are to be compared across groups, the rows should contain the characteristics and the columns should contain the groups. This is due to the fact that for our human mind it is easier to compare measures between columns than between rows. A great amount of information can be split to several tables in order to enhance

| Group | Frequency (in thousands) | Proportion from total |
|---|---:|---:|
| 1 | 13,2 | 47,1 |
| 2 | 8,5 | 30,4 |
| 3 | 3,8 | 13,6 |
| 4 | 2,5 | 8,9 |
| Total | 28 | 100 |

**Table 3** – An Example of a Frequency Table With Column Sums

readability. If this is for theoretical reasons not possible, long tables can be grouped by certain topics. It may even be helpful to include empty rows as structuring element. In general, tables should be structured as simple as possible, i.e., they should contain as little information and as few lines as possible. Also, try not to change your style of formatting within one document.

**Further reading**

- Chapman, C., & McDonnell Feit, E. (2015): R for Marketing Research and Analytics. Springer. *(Chapters 3.2 & 4.5, examples with R)*
- Handl, A. (2002): Multivariate Analysemethoden. Berlin: Springer. *(Chapter 2)*
- Schlittgen, R. (2003): Einführung in die Statistik. München: Oldenbourg. *(Chapter 2, detailed introduction for statistics-newbies)*
- Wickham, H., & Grolemund, G. (2017). R for Data Science. *(Containts several chapters about `ggplot2`)* Available under r4ds.had.co.nz/

## Exercises

In this exercise we will work with the `trees` data from the `data` package. You don't need to load it since it is already installed and preloaded, hence you can print the dataset directly to the console.

```
head(trees)
```

```
##   Girth Height Volume
## 1   8.3     70   10.3
## 2   8.6     65   10.3
## 3   8.8     63   10.2
## 4  10.5     72   16.4
## 5  10.7     81   18.8
## 6  10.8     83   19.7
```

You see the `Girth`, `Height` and `Volume` for 31 different black cherry trees. Now it's your turn!

1. Calculate the mean `Girth`, `Height` and `Volume` for the cherry trees. Then print the quantiles for the variable `Girth` to the console.
2. Using `ggplot2` create a scatter plot from the variables `Girth` and `Height`. Then add a linear smoothing line.
3. Create a QQ plot for the `Volume` variable.

### Solutions

1.

```
mean(trees$Girth)
```

```
## [1] 13.24839
```

```
mean(trees$Height)
```

```
## [1] 76
```

```
mean(trees$Volume)
```

```
## [1] 30.17097
```

```
quantile(trees$Girth)
```

```
##    0%   25%   50%   75%  100%
##  8.30 11.05 12.90 15.25 20.60
```

With dplyr, you can display the three means with one line of code:

```
trees %>% summarise_at(vars(Girth, Height, Volume), mean)
```

```
##      Girth Height   Volume
## 1 13.24839     76 30.17097
```

2.

```
library(ggplot2)
ggplot(trees, aes(Girth, Height)) +
  geom_point() +
  geom_smooth(method = 'lm')
```

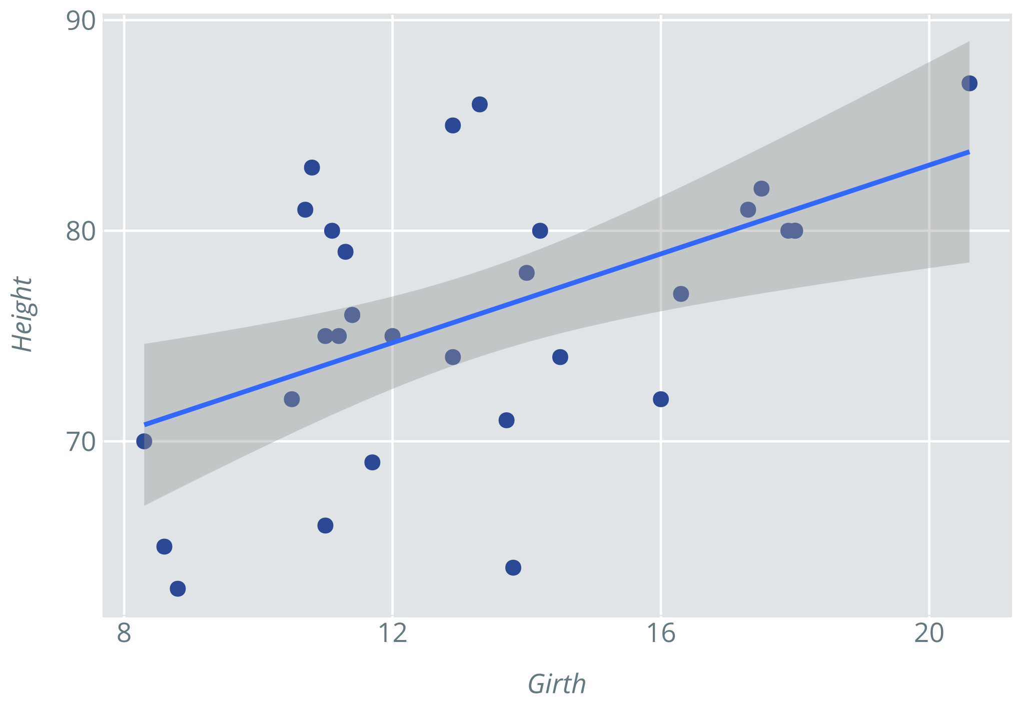


**Figure 23** – Scatter plot for `Girth` and `Height`

3.

```r
ggplot(trees, aes(sample = Volume)) +
  stat_qq()
```

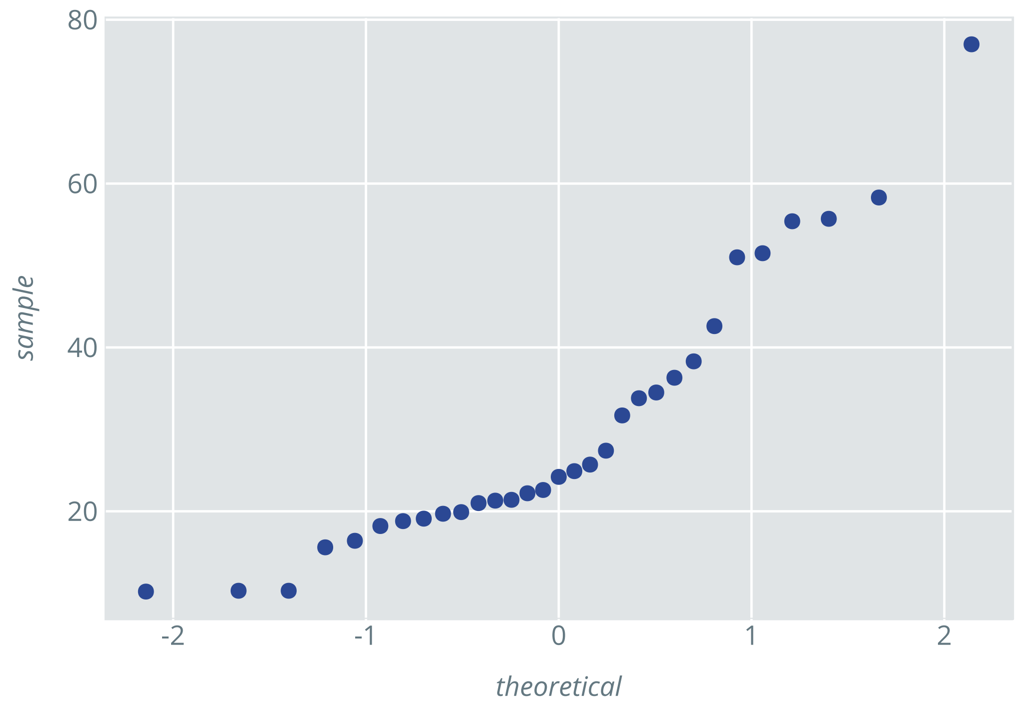


**Figure 24** – QQ plot for `Volume`

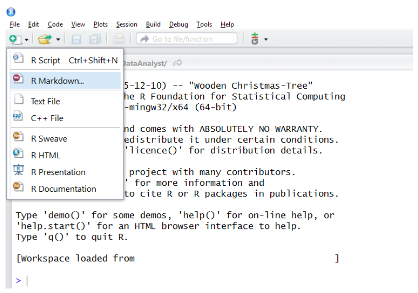
# 7 Reporting with R Markdown

In this section we are mainly going to present *R Markdown*. It lets you create and convert dynamic reports in `R` using the Markdown language. Additionally, you can embed `R` Code into reports by using the package `knitr`[32] which is the successor of `Sweave`[33]. The beauty of *R Markdown* and `knitr` is that writing is fairly simple but the output can be surprisingly complex. In particular you have the choice whether your output should be an automated report, beamer slides, interactive graphics, embedded shiny apps.... the sky is the limit[34]! The most common formats, when writing a report, are HTML, PDF or Word. *R Markdown* uses the core of `Markdown` which converts plain text to HTML[35].

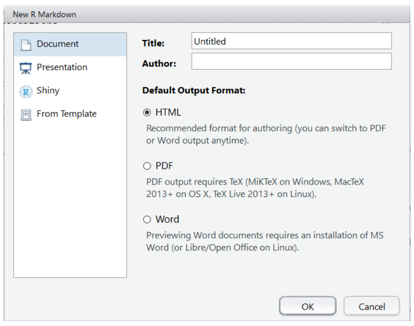
Behind the scenes the open source converter `Pandoc` is running. It handles Markdown, LaTex, HTML, Office Open XML and many more. If you choose PDF as output, additionally a Tex compiler is used.

And now, let's get started at the very beginning. The installation of R Markdown is done by:

```
install.packages("rmarkdown")
```



```
5-12-10) -- "Wooden Christmas-Tree"
he R Foundation for Statistical Computing
-mingw32/x64 (64-bit)

nd comes with ABSOLUTELY NO WARRANTY.
edistribute it under certain conditions.
'licence()' for distribution details.

 project with many contributors.
' for more information and
to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from                                    ]

> |
```



**Title:** Untitled
**Author:**

**Default Output Format:**

[✓ HTML]
Recommended format for authoring (you can switch to PDF or Word output anytime).

PDF
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

Word
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

**Figure 25** – How to create a new R Markdown file (`.rmd`) and select the title, author and output format

[32] Yihui Xie (2016). knitr: A General-Purpose Package for Dynamic Report Generation in R. R package version 1.12.

[33] `Sweave` converts text files into latex which in turn have to be converted to PDF by using the `texi2pdf` function from the `tools` package.

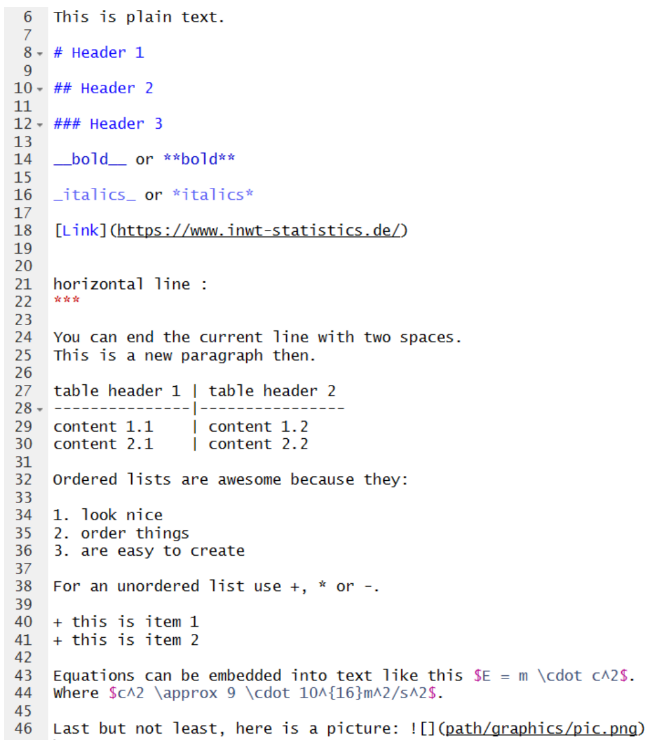
[34] See the following website for more information: http://rmarkdown.rstudio.com/index.html.

[35] "Markdown is a text-to-HTML conversion tool for web writers. Markdown allows you to write using an easy-to-read, easy-to-write plain text format, then convert it to structurally valid XHTML (or HTML). Thus, Markdown is two things: (1) a plain text formatting syntax; and (2) a software tool, written in Perl, that converts the plain text formatting to HTML." *Source:* The Daring Fireball by John Gruber (http://daringfireball.net/projects/markdown/

## 7.1 Text

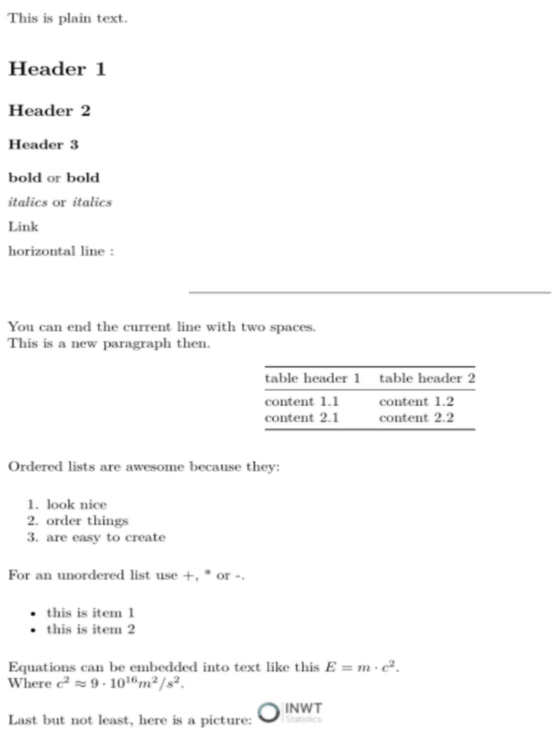
Some examples for the Markdown syntax and the respective output are:

**Syntax:**



```
This is plain text.

# Header 1

## Header 2

### Header 3

__bold__ or **bold**

_italics_ or *italics*

[Link](https://www.inwt-statistics.de/)


horizontal line :
***

You can end the current line with two spaces.  
This is a new paragraph then.

table header 1 | table header 2
---------------|---------------
content 1.1    | content 1.2
content 2.1    | content 2.2

Ordered lists are awesome because they:

1. look nice
2. order things
3. are easy to create

For an unordered list use +, * or -.

+ this is item 1
+ this is item 2

Equations can be embedded into text like this $E = m \cdot c^2$.
Where $c^2 \approx 9 \cdot 10^{16}m^2/s^2$.

Last but not least, here is a picture: ![](path/graphics/pic.png)
```

**Output:**



This is plain text.

Header 1

Header 2

Header 3

**bold** or **bold**

*italics* or *italics*

Link

horizontal line :

---

You can end the current line with two spaces.
This is a new paragraph then.

| table header 1 | table header 2 |
|---|---|
| content 1.1 | content 1.2 |
| content 2.1 | content 2.2 |

Ordered lists are awesome because they:

1. look nice
2. order things
3. are easy to create

For an unordered list use +, * or -.

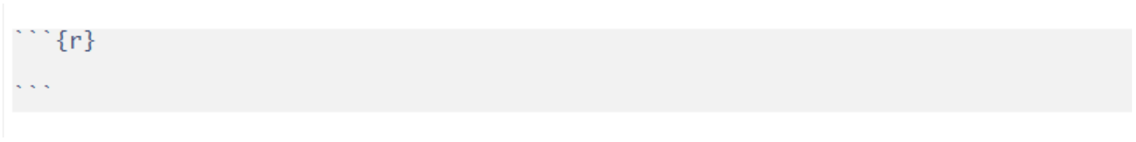
- this is item 1
- this is item 2

Equations can be embedded into text like this $E = m \cdot c^2$.
Where $c^2 \approx 9 \cdot 10^{16} m^2/s^2$.

Last but not least, here is a picture: INWT

## 7.2 Code Chunks

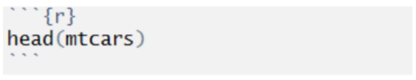
The package `knitr` allows us to embed `R` code into our R Markdown file. To insert a new code chunk, press `Strg+Alt+I` or click *Insert* ⟶ *R*. A code chunk always begins with ` ```{r} ` and ends with ` ``` `.



~~~
```{r}

```
~~~

So here is a simple example for printing the head of the `mtcars` dataset.

**Code Chunk:**



~~~
```{r}
head(mtcars)
```
~~~

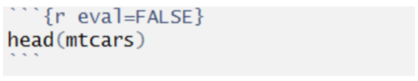
**Output:**

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

If you'd like only the R code to be shown in the report without being executed, you can set the chunk option `eval = FALSE`.

**Code Chunk:**



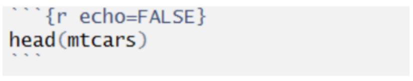
~~~
```{r eval=FALSE}
head(mtcars)
```
~~~

**Output:**

```
head(mtcars)
```

Or the other way round, you might want to display the output of the code but not the code itself. Then set `echo = FALSE`.

**Code Chunk:**



~~~
```{r echo=FALSE}
head(mtcars)
```
~~~

**Output:**

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

If you want the code to be executed, but you don't want to see the results, set the option `results = "hide"`.

If there are more than one chunk option you wish to specify, you can simply comma-separate them.

There are many more quite useful chunk options: You can find a selection in the table below. Further information about chunk options can be found here: http://yihui.name/knitr/options/#chunk_options. *Note:* When you specify chunk options, don't insert line breaks!

| Chunk Option (default, class) | Description |
|---|---|
| `eval` (TRUE, logical/numeric) | Whether the code chunk should be evaluated. Specific lines can be chosen with a numeric vector, i.e., line 1 to 3: `eval = (1:3)` or if you'd like only the 7th and the 10th line not to be evaluated: `eval = -c(7, 10)`. |
| `echo` (TRUE, logical/numeric) | Whether the code should be shown in the output file. Again you are allowed to pick specific code lines (not) to be shown. |
| `warning` (TRUE, logical) | Whether to show warning messages in the output file as you usually receive them in the R console. |
| `error` (TRUE, logical) | Whether to show error messages in the output file as you usually receive them in the R console. |
| `message` (TRUE, logical) | Whether to show messages. |
| `cache` (FALSE, logical) | Whether you want your code chunk to be cached. |

It is often the case that reports have more than 2 or 3 code chunks and of course we don't want to waste our time typing the same chunk options again and again. So what we can do is to change the global options. For example, if in our report we don't want any warnings and messages to be shown, we simply switch them off globally[36].

~~~
```{r}
knitr::opts_chunk$set(warning = FALSE, message = FALSE)
```
~~~

This makes life much easier. But there's even better news: you can include a report in your report! All you need to do is to give the name of another report to the chunk option `child`, then the report is being inserted just where the chunk option is specified.

~~~
```{r child='AnotherReport.Rmd'}

```
~~~

You can name your chunks by passing the name as the first argument (followed by the other options): ` ```{r chunkName, someOption = TRUE} `

## Further Reading

- Wickham, H., & Grolemund, G. (2017). R for Data Science. *(Part V)* Available under r4ds.had.co.nz/
- Xie, Y. (2015): Dynamic Documents with R and knitr. CRC press.

[36]Global options are overwritten by local options specified for a certain chunk.

**Exercises**

Hands on! Now build your own R Markdown report. You can choose on your own if you'd like your output to be a pdf or a html file. The solutions can be found in the next section, but try to do the exercises on your own. Don't forget to give meaningful names to your code chunks.

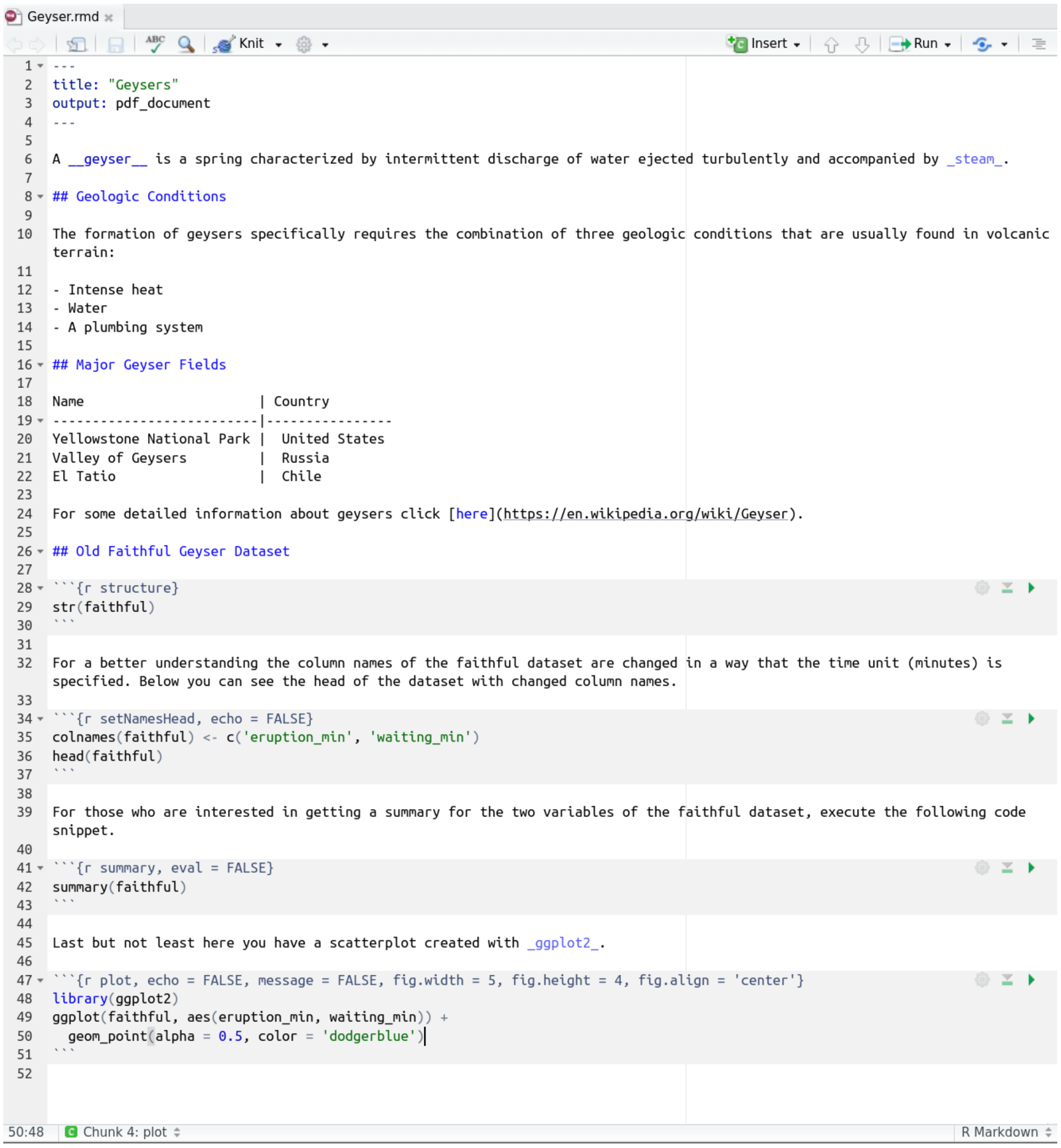
First, create a new report and name it `Geyser.rmd`, give it the title "Geysers".

1. Copy the following sentence: **A geyser is a spring characterized by intermittent discharge of water ejected turbulently and accompanied by steam.** Bold-emphasize the word "geyser" and italics-emphasize the word "steam".
2. Make a header called **Geologic Conditions**.
3. Write **The formation of geysers specifically requires the combination of three geologic conditions that are usually found in volcanic terrain:** and create an unnumbered list with the following features: **Intense heat, Water, A plumbing system**.
4. Make a new header **Major Geyser Fields** and then create a small table that looks like this:

| Name | Country |
|---|---|
| Yellowstone National Park | United States |
| Valley of Geysers | Russia |
| El Tatio | Chile |

5. Now create a link to the Wikipedia article about geysers and embed it into a short sentence (Example: For some detailed information about geysers click *here*).
6. Make another header called **Old Faithful Geyser Dataset**.
7. Insert your first code chunk that returns the dataset's structure (the dataset is called `faithful`).
8. The variable `eruptions` refers to the duration of each eruption in minutes and `waiting` is the waiting time to the next eruption in minutes. Change the column names to `eruption_min` and `waiting_min` and store the data in a new dataset called `geyser`. Print the head of the new dataset only (hide the code), but describe what you're doing.
9. Provide the readers of your report with the code to get the summary for both of the variables. But don't evaluate the code snippet.
10. The next code chunk should create a scatterplot from the `faithful` data. This time don't include the source code in the output file. Hint: The easiest way to create a scatterplot of `x` and `y` is `plot(x, y)`. But if you already know `ggplot2`, use it to make the plot looking nice. Don't show the message `Loading required package: ggplot2` either.
11. Find out how to change the size and alignment of the scatterplot created in the last exercise.

## Solutions



~~~
---
title: "Geysers"
output: pdf_document
---

A __geyser__ is a spring characterized by intermittent discharge of water ejected turbulently and accompanied by _steam_.

## Geologic Conditions

The formation of geysers specifically requires the combination of three geologic conditions that are usually found in volcanic terrain:

- Intense heat
- Water
- A plumbing system

## Major Geyser Fields

Name                      | Country
--------------------------|----------------
Yellowstone National Park |  United States
Valley of Geysers         |  Russia
El Tatio                  |  Chile

For some detailed information about geysers click [here](https://en.wikipedia.org/wiki/Geyser).

## Old Faithful Geyser Dataset

```{r structure}
str(faithful)
~~~

For a better understanding the column names of the faithful dataset are changed in a way that the time unit (minutes) is specified. Below you can see the head of the dataset with changed column names.

```{r setNamesHead, echo = FALSE}
colnames(faithful) <- c('eruption_min', 'waiting_min')
head(faithful)
```

For those who are interested in getting a summary for the two variables of the faithful dataset, execute the following code snippet.

```{r summary, eval = FALSE}
summary(faithful)
```

Last but not least here you have a scatterplot created with _ggplot2_.

```{r plot, echo = FALSE, message = FALSE, fig.width = 5, fig.height = 4, fig.align = 'center'}
library(ggplot2)
ggplot(faithful, aes(eruption_min, waiting_min)) +
  geom_point(alpha = 0.5, color = 'dodgerblue')
```
```
```

# 8 Functions

Functions in R are objects of class function and have the following structure:

```
someFunction <- function( <arguments> ) {
  <expression>
}
```

They are "first class objects", which means that they can be treated much like any other R object. For example,

- functions can be passed as arguments to other functions,
- functions can be returned by other functions,
- functions can be stored in a list, and
- functions can be nested, so that a function can be defined inside of another function.

## 8.1 Why functions?

Every time you do the same thing (or similar things) multiple times, you are well-advised to write a function. They make the code easier to maintain and less error-prone. The alternative would be copy and paste, but the following example shows why this is not the best solution.

Imagine a script where the mean and the maximum are printed for several vectors in order to check some previous computations. For a better overview, they are rounded to three resp. two decimals:

```
[...]

round(mean(x), 3)
round(max(x), 2)

round(mean(y), 3)
round(max(y), 2)

[...]

round(mean(z), 3)
round(max(x), 2)

[...]
```

While working with this script, you realize that you need only two decimals for the mean, or you want to compute the minimum as well... Do you see the point? Copy and paste is inelegant and dangerous for several reasons:

- It's hard to maintain: Each time you want to change something, you will have to change it in many lines of your code. This takes much time.
- Your script becomes very long.

- It's easy to overlook tiny errors... Did you realize that in the section for vector `z`, the maximum was computed for `x` instead of `z`? This may have happened because the author forgot to adjust this line after copying it from above.

Let's solve these issues with a function! We call it `mySummary()`. The input vector is just called `a`.

```r
mySummary <- function(a) {
  print(round(mean(a), 3))
  print(round(max(a), 2))
}

# Apply this function to vectors x, y, and z:
mySummary(x)
mySummary(y)
mySummary(z)
```

Now you can change the number of decimal places for the mean by just changing the respective number in the function. If you want to add the minimum, you just need to add a single line to the function. And if you apply the function more often, you will save many lines of code.

## 8.2 Components of a function

A function in `R` is an object of class `function` and contains three components: the body (access with `body()`), the formals or arguments (access with `formals()`), and the environment which determines how variables are found inside the function (access with `environment()`). Printing a function to the console shows all three components. If the environment is not specified, the global environment is the environment of the function.

To get an understandig of this, try to define a function called `f` with a single argument `x` which simply returns the value that is given as argument. First you can look at how the function is built by simply typing its name. Then access all three components of `f` one after the other.

```r
f <- function(x) {
  return(x)
}
```

```r
formals(f)
```

```
## $x
```

```r
body(f)
```

```
## {
##     return(x)
## }
```

```r
environment(f)
```

```
## <environment: R_GlobalEnv>
```

Note that functions can have a `return()` statement which indicates the object to be returned. This is, however, not necessary. In order to avoid redundant code, the use of return statements can be restricted to cases where a function call is stopped before the function's code is completely executed. Without a return statement, the function will return the last expression that was evaluated.

Some functions return invisible objects, for example the assignment operator <-. If you want to make invisible objects visible, you can put parentheses around the expression:

```
(x <- 5)
```

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

## 8.3 Function arguments

The formal arguments are the named arguments included in the function definition. These arguments can be matched by exact, positional and partial matching. Note that not every function call in `R` makes use of all the formal arguments. In fact, they can be missing or might have default values. It is possible to give arguments in terms of other arguments or in terms of values inside the function. The so-called actual or calling arguments are the arguments that are actually used in a function call.

As mentioned earlier, a `R` function's arguments can be matched `positionally` or by name. Long argument names can be abbreviated as long as the abbreviations are not ambiguous, i.e., might point to two different arguments. So the following calls to the function `sd()`, which returns the standard deviation of a variable, are all equivalent, but there are better and worse ways to write them:

```
# Sample from standard normal distribution
mydata <- rnorm(100)

# Compute standard deviation
sd(mydata)

sd(x = mydata)

sd(mydata, na.rm = FALSE)

# Ok, but unusual order of arguments
sd(na.rm = FALSE, x = mydata)

# Better to use unnamed arguments before named arguments
sd(mydata, na.rm = FALSE)

# Syntactically fine, but we prefer FALSE to be spelled out
sd(mydata, na.rm = F)
```

It is not recommended to mess around with the order of the arguments too much, since this can lead to confusion.

Function arguments can also be *partially* matched, which is a potential cause for errors and confusion. It can be useful for interactive work, though. The order of operations when given an argument is:

1. Check for exact match for a named argument,
2. Check for a partial match, or
3. Check for a positional match.

Note that in the following example, we can omit the curly brackets and the `return()` statement. This brings the advantage of fewer lines of code for simple functions. When you have more complex functions, you should stick to curly brackets though. It adds more structure to a function.

```
# Arguments in terms of other arguments:
someFunction <- function(x, y = x + 1) x + y

someFunction(2)
```

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

```

# Arguments in terms of values inside the function:
someOtherFunction <- function(x, y = z + 1) {
  z <- x + 1
  x + y + z
}

someOtherFunction(2)
```

```
## [1] 9
```

Defining default values in terms of values defined inside the function itself is usually bad practice since it is hard to understand the function call without reading the code of the function itself. The following function `myplot()` uses the argument `l` for “line” as default plot type. Additionally, it is possible to use the argument `...` to pass arguments on to other function calls. Generic functions use `...` so that extra arguments can be passed to methods (more on this later). In this case, the dots are used in the call to the `myplot()` function. This indicates that there are other arguments that can be passed to the `plot()` function inside of the function body.

```
myplot <- function(x, y, type = "l", ...) {
  plot(x, y, type = type, ...)
}
```

The `...` argument is also necessary when the number of arguments passed to the function cannot be known in advance. For the `sprintf()` function, for example, the number of variable values to be inserted into, let’s say, a character vector can vary from call to call. Similarly, the `cat()` function can take an arbitrary amount of `R` objects.

```
args(sprintf)
```

```
## function (fmt, ...)
## NULL
```

```
args(cat)
```

```
## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## NULL
```

One catch with `...` is that any arguments that appear after `...` on the argument list must be named explicitly and cannot be partially matched.

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed. The following function `f` never actually uses the argument `b`, so calling `f(2)` will not produce an error because 2 gets positionally matched to `a`.

```
f <- function(a, b) {
  a ^ 2
}

f(2)
```

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

```
# another example
f <- function(a, b) {
  print(a)
  print(b)
}

f(45)
```

```
## [1] 45
```

```
## Error in print(b): argument "b" is missing, with no default
```

Note that "45" got printed first and before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)`, it had to throw an error. Even if you specified `b` outside the function but didn't pass it as argument to the function, `b` could not be printed either.

The reason that arguments can be defined in terms of other arguments is the concept of lazy evaluation. Expressions in a function are only evaluated when they are needed, which means that before `R` tries to evaluate the values of the argument a new environment is created in which the evaluation takes place.

So let's see what happens if we execute the following code:

```
x <- 1
someFunction <- function(x, y = x + 1) x + y
someFunction(x = 2)
```

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

Well, even though we specified x to be 1 outside of the function, 2 + 2 + 1 = 5 is returned instead of 1 + 1 + 1 = 3. This is because a function can be seen to create its own environment. So the object outside the function (in the global environment) named x is masked inside the function. Hence R can only find the object named x which is the value of the argument x. Additionally, at the time when the value for y is needed (when y is evaluated), x is already part of the environment.

## 8.4 Replacement functions

Apart from the functions we just learned about, there is a thing called replacement functions. A replacement function acts (syntactically) as if it modifies its argument. Internally it is an 'ordinary' function and distinguished with a special naming. It takes its argument, manipulates and returns it[37].

One rather common replacement function is the names() function. Take for example the R-intrinsic iris data frame. If you call names() on it, you get a character vector with the column names.

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

The function can as well be used to alter the column names. So let's change the names to lowerCamelCase:

```
names(iris) <- c("sepalLength", "sepalWidth",
                 "petalLength", "petalWidth",
                 "species")
names(iris)
```

```
## [1] "sepalLength" "sepalWidth"  "petalLength" "petalWidth"  "species"
```

And really, we changed the column names. So when evaluating the expressions first of all R will notice that the left hand side is not a simple name but a function call. It then searches for a function called 'names<-' and one called 'names'. This implies that there are two function definitions for 'names', names and names<-:

names takes one argument, in the case presented above the iris data frame.

```
str(names)
```

```
## function (x)
```

```
names
```

```
## function (x)  .Primitive("names")
```

[37]This is not quite true. Actually a modified copy is returned. This can be seen by checking the memory adress of the object before and after the replacement function is applied.

names<- by construction takes two arguments.

```
str(`names<-`)
```

```
## function (x, value)
```

```
`names<-`
```

```
## function (x, value)  .Primitive("names<-")
```

Consider the following example: We are looking at the vector z containing the integers 1 to 20. Now let's say we are not interested in the values above a certain cutoff value. Hence we want to design a function that sets these values to the cutoff. This can be nicely achieved using a replacement function:

```
# Defining a vector z
z <- 1:20

# Defining the replacement function "cutoff<-"
# which replaces all values above a certain value with that certain value
"cutoff<-" <- function(x, value) {
  x[x > value] <- value
  x
}

# calling on it
cutoff(z) <- 13
z
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 13 13 13 13 13 13 13
```

So obviously it worked. Since replacement functions are simply functions, the usual function call is also valid. However, when using special names, the function name must be referred to using single quotes. So here is what actually happened with our `cutoff()` function:

```
z <- "cutoff<-"(x = z, value = 13)
```

Note that R only knows one way to interpret a function call and that is: `functionName(arguments)`. When evaluating replacement functions, the initial expression:
`replacementFunction(x) <- value`
is rewritten as:
`x <- 'replacementFunction<-'(x, value).`

### 8.5 Infix functions

Most of the time, functions are used as 'prefix' operators. However, binary operators like '+', '–', '==' and even '<–' are also functions. They are called 'infix' operators since they are set inbetween two operands. If an infix operator is already reserved but you still want to create a new function with that operator, you have to use double quotes to be able to assign the function definition to this variable name – see the example.

Usual Usage:

```r
1 + 2
```

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

New definition:

```r
"+" <- function(x, y) x - y
```

Let's try:

```r
1 + 2
```

```
## [1] -1
```

```r
rm("+")
```

Wow, of course the last result is not what you would expect if you didn't know about our previous definition. This is why we should avoid to overwrite established functions and operators. Package creation and namespaces serve as good tool to avoid confusion about function definitions.

You can create your own infix operator using the symbol '%'. There are already predefined functions in R using this structure: `%*%`, `%/%`, etc. But let's see a more useful example than overriding the functionality of the "+"-sign: For example, creating an operator for pasting strings together.

```r
"%+%" <- function(x, y) paste(x, y)
"some" %+% "string"
```

```
## [1] "some string"
```

The naming of functions in this context is more flexible: You are allowed to use all symbols except `%`.

```r
"%paste%" <- function(x, y, ...) paste(x, y, ...)
"new" %paste% "string"
```

```
## [1] "new string"
```

If you want to refer to infix operators without using them between operands, you can use single quotes: ’ to call on them. This might be useful for calling the help page. Another usecase is the following way to call on a function, which is equivalent to the way we have seen before.

Adding numbers:

```r
# Common syntax:
1 + 2
```

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

```r

# Rearranged:
'+'(1, 2)
```

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

Pasting strings:

```
# Common syntax:
"new" %paste% "string"
```

```
## [1] "new string"
```

```
# Rearranged:
'%paste%'("new", "string")
```

```
## [1] "new string"
```

You should, however, not switch the way you are calling on infix operators within one project in order to avoid confusion and retain readability of your code.

## 8.6 Functions inside other functions

In R, functions can be defined anywhere and in any environment. Function definitions inside of other functions will determine where a function is available. With functions, the scoping rules (see Section 9 below) apply as for any other object. If function a is defined inside function b, then function a will only be available during the evaluation of function b.

```
b <- function(x) {
  a <- function(y) y ^ 2
  a(x)
}

b(2)
```

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

```
# global environment
exists("a")
```

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

### Further reading

- Chambers, J. M. (2008): Software for Data Analysis. New York: Springer. *(Chapter 3; in-depth, theoretical introduction)*
- Chapman, C., & McDonnell Feit, E. (2015): R for Marketing Research and Analytics. Springer. *(Chapter 2.7, short applied introduction)*
- Wickham, H. (2017): Advanced R. *(Chapter 6)* Available under adv-r.had.co.nz/Functions.html
- Wickham, H., & Grolemund, G. (2017). R for Data Science. *(Chapter 19)* Available under r4ds.had.co.nz/functions.html

### Exercises

Ok, that's enough input for the beginning: It's your turn now. In the following section, we present some exercises for the topics you just learned. You find the solutions at the end of the section.

### Functions

**1.** For each of the following functions, predict the return value and think about meaningful function names instead of `f1, f2, ...`. If you do not know what the functions `sum, length` and `order` do, have a look at the help files. Do the computation to check if your predictions are correct.

**a)**

```r
f1 <- function(r) pi * r ^ 2
f1(r = 1)
```

**b)**

```r
f2 <- function(r) 2 * pi * r
f2(r = 1)
```

**c)**

```r
f3 <- function(x) sum(x) / length(x)
f3(x = c(1, 1, 3, 3))
```

**d)**

```r
f4 <- function(x) sum((x - f3(x)) ^ 2)
f4(x = c(1, 1, 3, 3))
```

**e)**

```r
f5 <- function(x) x[order(x)]
f5(x = c("c", "b", "a"))
```

**f)**

```r
f6 <- function(x, ...) x[order(x, ...)]
f6(x = c("a", "b", "c"),
   decreasing = TRUE)
```

**2.** Write a function `meanVarSdSe` that takes a numeric vector `x` as argument. The function should return a named vector that contains the mean, the variance, the standard deviation `sd` and the standard error `se` of `x`. The standard error is defined as

$$se(x) = \frac{sd(x)}{\sqrt{\#x}}, \qquad (1)$$

where `#x` denotes the cardinality, i.e., the number of elements contained in `x`.

**a)** The code should have the following structure

```r
meanVarSdSe <- function(x) {
  ### your code
}
```

and return a named vector according to

```r
x <- 1:100
meanVarSdSe(x)
```

```
##       mean        var         sd         se
##  50.500000 841.666667  29.011492   2.901149
```

You can use the functions `mean`, `var`, `sd` and `length`. Check the help files of these functions for further arguments that can be used optionally.

**b)** Look at the following code sequence. What result do you expect?

```r
x <- c(NA, 1:100)
meanVarSdSe(x)
```

Now run the code. Explain the result. Extend the function definition of `meanVarSdSe` with the argument `...`, as is illustrated as follows:

```r
meanVarSdSe <- function(x, ...) {
  ## your code
}
```

so that the `na.rm = TRUE` argument can be passed optionally to the functions `mean`, `var` and `sd`.

What is the correct value for $\#x$ in the case of missing values? Use `sum(!is.na(x))` as denominator in equation (1). Read the help page for the function `is.na()`. The optimized function should return:

```r
x <- 1:100
meanVarSdSe(c(x, NA), na.rm = TRUE)
```

```
##       mean        var         sd         se
##  50.500000 841.666667  29.011492   2.901149
```

**3.** See what happens if after arguments marked with `...`, you want to set another argument, but forget to name it correctly:

```r
cat(letters[1:10], fil = TRUE, labels = paste0("{", 1:10, "}:"))
```

What do you think happened?

**4.** Remember what happened when we called on the following function:

```r
f <- function(a, b) {
  print(a)
  print(b)
}
```

Now rebuild the function such that it does not throw an error when only one argument is passed.

**5.** Sometimes it might be useful to specify a function's arguments outside the function, e.g., when they are used in several calls and you don't want to copy and paste them over and over again. Look at the help page of the function `do.call()` and rewrite the following function call specifying its arguments beforehand.

```
mydata <- 1:20
sd(mydata, na.rm = FALSE)
```

### Replacement functions

**6.** Consider the following code and output:

```
x <- 1:10
replacementFunction(x) <- -1
x
```

```
##  [1]  1 -1  3  4  5  6  7  8  9 10
```

```
y <- 40:50
replacementFunction(y) <- -1
y
```

```
##  [1] 40 -1 42 43 44 45 46 47 48 49 50
```

What would you think: How was the replacement function defined?

**7.** It is often very useful to combine replacement and subsetting. Expressions like the following are also valid:

```
x <- c(a = 1, b = 2, c = 3)
names(x)[2] <- "two"
x
```

```
##   a two   c
##   1   2   3
```

What do you think this is internally turned into?

### Infix functions

**8.** Write an infix function `%and%` that behaves like the logical operator. Use only the `ifelse` control structure (check `help("ifelse")` to see what the `ifelse` control structure does) and the function '==':

```
TRUE %and% TRUE
```

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

```
FALSE %and% TRUE
```

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

```
TRUE %and% FALSE
```

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

```
FALSE %and% FALSE
```

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

**9.** We have seen two ways of conducting simple operations in R. Let's translate the one into the other.
How can the following operation be expressed differently?

```
'/'(12,6)
```

**10.** How else can you write the following assignment?

```
x <- 1:5
```

**11.** How else can you write the following subsetting?

```
x[5]
```

**Solutions**

**Functions**

**1.** In the following, we present some meaningful function names. Note that we like the naming convention `lowerCamelCase`.You can check the return value yourself by executing the code.

**a)** areaCircle

**b)** circumferenceCircle

**c)** averageVector

**d)** sumOfSquares

**e)** sortVectorAscend

**f)** sortVector

**2.** Function `meanVarSdSe` for calculating mean, variance, standard deviation and standard error and returning them in form of a a named vector.

**a)** Function code:

```
meanVarSdSe <- function(x) {
  n <- length(x)
  c(mean = mean(x),
    var = var(x),
    sd = sd(x),
    se = sd(x) / sqrt(n))
}
```

**b)** Without further specifying an option for handling missing values, the functions used "return missing values" in the presence of `NA`s. Hence, we have to extend the function such that the argument `na.rm = TRUE` can be passed through.

```
meanVarSdSe <- function(x, ...) {
  n <- length(na.omit(x))
  c(mean = mean(x, ...),
    var = var(x, ...),
    sd = sd(x, ...),
    se = sd(x, ...) / sqrt(sum(!is.na(x))))
}
```

```
x <- 1:100
meanVarSdSe(c(x, NA), na.rm = TRUE)
```

```
##       mean        var         sd         se
##  50.500000 841.666667  29.011492   2.901149
```

**3.** After `...`, partial matching does not work and thus the wrongly spelled argument `fil` was not identified as valid argument. Hence, its value `TRUE` was taken as additional element to be concatenated and printed. The succeeding argument `labels` was ignored. Setting things right, you can check what kind of result we wanted to achieve:

```
cat(letters[1:10], fill = TRUE, labels = paste0("{", 1:10, "}:"))
```

```
## {1}: a b c d e f g h i j
```

**4.** The function `f`, as it is defined above, requires two parameters to be printed: `a` and `b`. If only one argument is given, it throws an error when evaluated (lazy evaluation). This can be avoided by providing default values in the function definition:

```
f <- function(a = "a not specified", b = "b not specified") {
  print(a)
  print(b)
}
```

**5.** The function `do.call()` constructs a function call from a function name and a list of arguments. In this case, its first argument is `sd`, the function's name, and its second argument is `args`, a list of the arguments to be passed to the function `sd()`.

```
args <- list(mydata, na.rm=FALSE)
do.call(sd, args)
```

### Replacement functions

**6.** The function simply replaces the second element of the vector with the value assigned. It is defined as follows:

```
"replacementFunction<-" <- function(x, value) {
  x[2] <- value
  x
}
```

**7.** Replacement functions take their argument, copy and modify it and return the modified copy. Hence, the names of `x` are copied to the temporal object `tmp`. Afterwards, the second element is replaced via subsetting and the result is assigned back to the names of `x`:

```
x <- c(a = 1, b = 2, c = 3)
tmp <- names(x)
tmp[2] <- "two"
names(x) <- tmp
x
```

```
##   a two   c
##   1   2   3
```

### Infix functions

**8.** In general, we want to minimize calculation effort. So it makes sense here to start checking the condition `FALSE` first, in order to be able to interrupt evaluation as soon as possible and return `FALSE`.

```
"%and%" <- function(a, b) {
  ifelse(a == FALSE,
         return(FALSE),
         ifelse (b == TRUE,
                 return(TRUE),
                 FALSE)
  )
}
```

**9.** Using an infix operator: `12/6`

**10.** Changing from infix function to alternative notation: `'<-'(x, 1:5)`

**11.** Even the subsetting operator can be put up front: `'['(x,5)`

# 9 Scoping Rules

Scoping rules are the set of rules which define how R finds objects. They are basically governed by two concepts: lexical scoping and dynamic lookup. Lexical scoping determines *where* to look for values. The dynamic lookup determines *when* to look for them. The following aspects will be discussed in greater detail[38]:

- Environments,
- Name masking,
- Environments and functions,
- Dynamic lookup, and
- The search path.

## 9.1 Environments

An environment binds a set of names to values. Environments are objects and can have a name and be stored in lists etc. Although they are very similar to lists, there are three fundamental differences:

First, evironments have **reference** semantics – whenever you modify an environment, you modify every copy. What we use as objects are pointers to the actual values. Copying an environment means copying the pointer, not the value! Environments can be useful data structures, however, reference objects should only be used with great care. Typically you will only work implicitly with environments.

Environments are objects – you can find them in your workspace. And they are similar to lists. Elements of them can be accessed by using the dollar sign.

```
# Checking current environment
ls()
```

```
## [1] "objectsToKeep" "params"        "pathData"      "pathGraphics"
```

```
e <- new.env()
e$x <- 1
ls(e)
```

```
## [1] "x"
```

```
e$x
```

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

The second difference between environments and usual lists is that environments have **parents**: If an object is not found in an environment, R will continue to search in the parent, which is an environment and thus has a parent... When a parent is not explicitly set – which is certainly possible – the default is the environment in which it is created. If an object is needed but cannot be found inside an environment, R searches one environment

[38]This section is mainly based on H. Wickham 2015: Advanced R. CRC Press.

higher and sees if it can find the object there. The highest level is the empty environment which does not contain anything.

Third, every object in an environment must have a **name** and the names must be unique. Subsetting with position and logical vectors is not possible for environments – only with names.

For most objects in `R`, the 'copy-on-modify' semantics are applied, meaning there is no true modification possible, only replacements. Every time an object is modified, it is first copied, then modified, and then it replaces the existing object. Only in rare cases, the object is modified right away.

## 9.2 Name masking

Whenever there is more than one object with the same name, the object in the "closest" environment will be selected. An environment is closer if it appears earlier in the search path which can be seen via `search()`. First, we look at the function code of the `dim()`- function. Then we define our own `dim()` function, working somewhat differently.

```
dim # predefined function
```

```
## function (x)  .Primitive("dim")
```

```
dim <- function(x) x # defining a new function
```

Now we need a way to look at the places the functions can be found at. So we define the following function:

```
# finding environment of function
searchForFun <- function(funOfInterest) {
  loadedEnv <- search()
  objInEnv <- sapply(as.list(loadedEnv),
                     function(env, ...) {
                       exists(envir = as.environment(env),
                              ...)
                     },
                     x = funOfInterest, inherits = F)
  loadedEnv[objInEnv]
}
```

We extract two locations: The `workspace` and the *base* package.

```
# looking at environments of dim
searchForFun("dim")
```

```
## [1] ".GlobalEnv"   "package:base"
```

```
dim # workspace comes first
```

```
## function(x) x
```

```
base::dim # use colon in order to access package
```

```
## function (x)  .Primitive("dim")
```

Here, two functions with the same name can be found. Whenever you call the function `dim()`, `R` will use the version in your `workspace`. The function `dim()` in the *base* package is not overwritten, it is simply masked by the function in the `workspace`. If we want to explicitly specify the package where the function should be taken from, we can use the colon, e.g., `base::dim()`.

```
# Remove "wrong" dim function to avoid side effects
rm(dim)
```

## 9.3 Environments and functions

Every call to a function creates a temporal environment in which the function call is evaluated. This guarantees (potentially) that the evaluation does not affect the global environment – your workspace. It also determines how names are looked for inside a function (we only consider user-written functions). If not specified differently, the environment is the global environment plus what is passed into the function in form of formal arguments and objects created inside the function. Those formal arguments and objects are called “local” and are only available during the function call. Hence, the function does not mind how often you call it, and it cannot know if it was called before. Of course, you can assign the function call to an object and save the result of the function run.

If it is obvious that a function is being searched for (e.g., by calling `n()`), `R` searches as long as it finds a function, even though it might find an object called `n` before.

Here are some examples: Always first think about the answer theoretically, then check yourself.

What is the value of `x`?

```
x <- 1

someFunction <- function(x) {
  x <- x + 1
  x
}
```

And here is the solution:

```
someFunction(x)
```

```
## [1] 2
```

And what is the value of `y`?

```
x <- 1

otherFunction <- function(x) {
  y <- x + 1
  y
}
```

Solution:

```
otherFunction(x)
```

```
## [1] 2
```

What is the result of a call to `moreFunction()` below? What is the result of the second call to `moreFunction`? What kind of explanation can you think of?

```
moreFunction <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  print(a)
}
```

Let's see what happens:

```
moreFunction()
```

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

```
moreFunction()
```

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

## 9.4 Dynamic lookup

So far, we have looked at lexical scoping, which determines where to look for values. Now we examine when exactly values are searched for. The underlying principle is called "dynamic lookup". It says that when a function is created, `R` does not check right away if the variables which appear in the function are available. This does not happen before the function is run. This implies that a function can have different return values depending on the objects defined in the environment in which it is created and run. Thus, an important property of well-defined functions can be violated: self-containment. The property of self-containment is fulfilled if the return value of a function depends only on the input values (formal arguments). You should always write self-contained functions. If you don't, it is very easy to produce unexpected results.

The following function definition is valid since `R` will not try to find the value of `y` until `oneFunction()` is called.

```
# building a function using y which does not exist
oneFunction <- function(x) x + y
```

Depending on the value of `y`, the function `oneFunction()` will have a different return value:

```r
y <- 1
oneFunction(1)
```

```
## [1] 2
```

```r
y <- 2
oneFunction(1)
```

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

In this definition, `oneFunction()` is not self-contained, i.e., even though the value of `x` is given as input value, the definition of `y` is left to the global environment. Such behaviour should be avoided if possible.

## 9.5 The search path

Every evaluation in `R` is made in a specific environment. The workspace is the "global environment". It can be accessed using `globalenv()` or using the built-in object `.GlobalEnv`.

Try `ls(.GlobalEnv)` to see what is in your workspace. The names should correspond to what you see in `R`. The workspace typically contains the objects you initialized in your current session. The question is then how `R` can find the built-in functions (e.g., `dim()`, which is not in your workspace). To omit the explicit reference to a package from which we want to use a function, the search mechanism for environments is applied. Since the workspace (the global environment) is an environment, it has a parent. Typically this is the last package loaded, which itself is again an environment. The parent of the last package loaded is the package loaded before and the connection between the loaded packages is the search path. The search path is used to determine the order in which the loaded packages are searched to find a function or any other object. The "end" of the search path is *package:base* (`baseenv()`). The parent of *package:base* is the empty environment, `emptyenv()`, which itself has no parent, is empty and cannot contain any objects: This is where the search ends. The `search()` command returns the search path.

```r
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

```r
library(dplyr)

# loading a package puts it at the beginning of search path
search()
```

```
##  [1] ".GlobalEnv"        "package:dplyr"     "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

**Further reading**

- Chambers, J. M. (2008): Software for Data Analysis. New York: Springer. *(Chapter 5.3 explains environments)*
- Wickham, H. Advanced R (2017). *(Chapters 6 and 7)* Available under adv-r.had.co.nz/

### Exercises

#### Environments

**1.** Create a new environment and check which environment was set as its parent.

**2.** Now do the same and set the parent explicitly. How can you check whether it worked or not?

**3.** Imagine we have a dataset `x` with two observations and one variable `V1`. So let's use the function `nrow` to examine it further. What does it do? And how does the function code look like? Now let's define our own functions `dim` and `newNrow` – they probably act somewhat differently.

```
x <- data.frame(V1 = 1:2)
str(x)
```

```
## 'data.frame':    2 obs. of  1 variable:
##  $ V1: int  1 2
```

```
nrow # looking at function code
```

```
## function (x)
## dim(x)[1L]
## <bytecode: 0x00000000164f5af0>
## <environment: namespace:base>
```

```
# let's define our own functions
dim <- function(x) 1
newNrow <- function(x) dim(x)[1L]
```

What would you expect calling `nrow(x)` and `newNrow(x)`? What does actually happen? Why do you think is that the case?

**4.** Look at the following function. What is the return value of `f(10)`?

```
f <- function(x) {
  f <- function(x) {
    f <- function(x) {
      x ^2
    }
    f(x) + 1
  }
  f(x) * 2
}
f(10)
```

#### Functions and environments

**5.** What is the result of a call to `someFunction`? What happened?

```r
x <- 1
someFunction <- function() x
```

**6.** What is the result of a call to `someFunction`?

```r
rm(list = ls())
x <- 1
someFunction <- function() x * y
```

**7.** The same rules apply if you define functions inside other functions. So what do you expect as result when the following function is called?

```r
x <- 1

someFunction <- function() {
  y <- 2
  otherFunction <- function() {
    z <- 3
    c(x, y, z)
  }
  otherFunction()
}
```

INWT Statistics

**Solutions**

**Environments**

**1.** If we simply construct a new environment, it is built per default in the global environment:

```
f <- new.env()
f$x <- 1
parent.env(f)
```

```
## <environment: R_GlobalEnv>
```

**2.** We can alternatively set the parent explicitly. In order to check whether it worked, we can use the function `parent.env()`:

```
e <- new.env(parent = f)
e$y <- 2
parent.env(e)
```

```
## <environment: 0x0000000014af7728>
```

```
f
```

```
## <environment: 0x0000000014af7728>
```

**3.** This exercise is about functions and environments. First we examine the function `nrow()`:

```
# nrow environment
x <- data.frame(V1 = 1:2)
str(x)
```

```
## 'data.frame':    2 obs. of  1 variable:
##  $ V1: int  1 2
```

```

nrow # looking at function code
```

```
## function (x)
## dim(x)[1L]
## <bytecode: 0x00000000164f5af0>
## <environment: namespace:base>
```

We see that the function is located in the *base* package and, internally, it uses the function `dim()`. Our own functions are both defined in the global environment.

```
dim <- function(x) 1
newNrow <- function(x) dim(x)[1L]
```

If we now execute both `nrow` functions, we can check whether or not they use different `dim` functions. And in fact, they return different results. Obviously, the old `nrow()` function uses the `dim()` function from the *base* package and the `newNrow()` function uses the previously defined `dim` function. Using `search()` returns the order in which the packages and objects are searched. Since our `newNrow()` function lives in the global environment, it uses the `dim()` function living in the global environment as well. On the contrary, the `nrow()` function living in the *base* package uses the `dim()` function living in the same enviroment.

```
nrow(x)
```

```
## [1] 2
```

```
newNrow(x)
```

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

```
ls(all.names = TRUE)
```

```
##  [1] ".Random.seed"  "dim"           "e"             "f"
##  [5] "newNrow"       "objectsToKeep" "params"        "pathData"
##  [9] "pathGraphics"  "x"
```

**4.** The functions are evaluated from the inside to the outside. Hence, the result is 202.

### Functions and environments

**5.** Since the function `someFunction()` does not take any parameter, it always returns `x` as defined in the workspace.

**6.** Since `y` is not defined, the call to `someFunction()` returns an error.

```
x <- 1
someFunction <- function() x * y
someFunction()
## Error in someFunction(): object 'y' not found
```

**7.** Since `R` uses the concept of lazy evaluation, the parameters `x`, `y`, and `z` are not needed until the function `someFunction()` is called. Since `x` is defined in the working directory and `y` and `z` are defined inside of the function, a call to `someFunction()` returns `1,2,3`.

```
# Remove "wrong" dim function to avoid side effects
rm(dim)
```

# 10 Control Structures

Control structures in R allow you to control the flow of an R program, depending on runtime conditions. You can find common structures in the table below.

| Structure | Use |
|---|---|
| `if, else` | test a condition |
| `for` | execute a loop a fixed number of times |
| `while` | execute a loop while a condition is true |
| `repeat` | execute an infinite loop |
| `break` | break the execution of a loop |
| `next` | skip an iteration of a loop |

Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions.

You should pay attention when using infinite loops, however. The best is to avoid them if possible, even if they are theoretically correct. For command-line interactive work and "advanced" code, the `*apply` functions are more useful and advisable than `for` or `while` loops. Programming iterative processes (i.e., processes where one computation step needs the result of a preceeding step), however, is not possible with `*apply` functions.

## 10.1 if else

The following expressions show two `if-else` structures, the second one being a bit more complex than the first one.

```
if (condition) {
  ## do something
} else {
  ## do something else
}
```

```
# or
if (condition1) {
  ## do something
} else if (condition2) {
  ## do something different
} else {
  ## do something even more different
}
```

Note that the else clause is not necessary.

The following statement checks whether `x` is greater than three. If so, the value 10 is assigned to `y`. If not, zero is assigned.

```
x <- 5

if (x > 3) {
  y <- 10
} else {
  y <- 0
}
```

### 10.2 switch

`switch` is very similar to `if-else`, but if there are many consecutive `else if ...` statements, `switch()` can be faster and easier to read. The first argument `EXPR` is a number or a character string. The remaining arguments are multiple alternatives.

If `EXPR` is a number, the alternatives are just numbered consecutively:

```
switch(EXPR = 1,
       "first alternative", "alternative number 2", "third alternative")
```

```
## [1] "first alternative"
```

```
switch(EXPR = 3,
       "first alternative", "alternative number 2", "third alternative")
```

```
## [1] "third alternative"
```

This is of course not very useful yet, because the `EXPR` is a constant number, so you could just type the result directly. It makes more sense to use it in a function and to pass an object as `EXPR` argument. Let's write a function which returns a character string according to a number:

```
mySwitchNum <- function(x) {
  switch(EXPR = x,
         "first alternative", "alternative number 2", "third alternative")
}
mySwitchNum(1)
```

```
## [1] "first alternative"
```

```
mySwitchNum(2L) # Integers are just turned into numerics.
```

```
## [1] "alternative number 2"
```

```
mySwitchNum(4) # There is no 4th alternative, so nothing is returned.
```

If `EXPR` is of class character, `switch` can work with named arguments. In the following, we write a function `mySwitchChar` which takes a dataframe and applies a method that you choose via `switch`. The first method prints the names, the second method returns the dimensions. The last alternative always covers all remaining cases. You don't need to provide this "residual category", but it often makes your code cleaner.

```
mySwitchChar <- function(method, data) {
  switch(EXPR = method,
         names = names(data),
         printDims = cat("Rows:", nrow(data), "\nCols:", ncol(data)),
         "Method not implemented!") # For all remaining cases
}

mySwitchChar("names", iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

```
mySwitchChar(2, iris) # You can still choose the method by position
```

```
## Rows: 150
## Cols: 5
```

```
mySwitchChar("computeSomeGreatModel", iris) # A method which was not defined
```

```
## [1] "Method not implemented!"
```

## 10.3 for

`for` loops take a loop variable and assign successive values from a sequence or a vector to it. `for()` loops are most commonly used for iterating over the elements of an object (list, vector, etc.). The following loop takes the variable `i` and assigns the values contained in the vector `1:5` to it successively in each iteration of the loop. At the end it exits. Note that the letter *i* is commonly used as looping variable. It could, however, be any other letter or word.

```
for (i in 1:5) {
  print(i)
}
```

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

Other forms of `for` looping are shown below. These four loops have the same behavior. They all return the elements of the vector `x` successively.

```
x <- c("a", "b", "c", "d")

for (i in 1:4) {
  print(x[i])
}
```

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

```
# looping over numeric indices
for (i in seq_along(x)) {
  print(x[i])
}

# looping over the elements
for (letter in x) {
  print(letter)
}

# same as above, just "letter" replaced by "i"
for (i in x) print(i)
```

Note that `for` loops can be nested. We will show you how in a second. However, it is not recommended to use nesting too excessively because it renders code unreadable. Additionally, `for` loops are rather slow in execution and, if possible, *`apply` structures should be preferred (see Section 11.1).

Anyways, first we generate a matrix with three rows and two columns. We then use nested `for` loops in order to subset each element of the matrix. Here, the inner `for` loop moves faster than the outer loop, i.e., the index of the column changes while the index of the row stays the same until the row ends.

```
x <- matrix(data = 1:6, ncol = 2, nrow = 3, byrow = TRUE)

for (i in seq_len(nrow(x))) {
  for (j in seq_len(ncol(x))) {
    print(x[i, j])
  }
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
```

A common thing to do when programming with `R` is to use `for` loops in order to dynamically build up a vector. Take, for example, a starting capital of 100 EUR. Now, your grandmother gives you 50 Euro for Christmas every year. You could use the following loop to create a vector containing your yearly capital for the next five years.

```
capital <- 100
```

```r
n <- 5
for (i in 1:n) {
  capital <- c(capital, capital[i - 1] + 50)
}
```

Of course, the nice thing is that you can specify for how many years you want to have your capital displayed. But from a programming point of view, this loop is not pretty at all. It is very inefficient in how the vector `capital` is filled. Every time the vector is overwritten, a copy has to be made, then the copy is modified and rewritten to `capital`. A faster way would be to specify the length of the result vector up front and then fill in the elements. Or to use a completely other structure...

## 10.4 while

`while` loops begin by testing a condition. If it is true, the loop body is executed. Once the loop body is executed, the condition is tested again. This is repeated until the condition is false – then the loop stops.

```r
number <- 0

while (number < 5) {
  print(number)
  number <- number + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

`while` loops can potentially result in infinite loops if not written properly. So use them with care! Of course it is possible to include more than one condition in the test.

### 10.4.1 Random walk

In the following – more complex – example, we are looking at a drunk person stepping out of a bar at a harbour. We imagine the person being so drunk that he randomly takes steps to the left or the right. This process corresponds to the so-called random walk, a mathematical model where each step is taken randomly, independent of the present status of the process. Since the bar is located directly at the pier, the person should not step further than three steps to the left (denoted as negative numbers) and five steps to the right (denoted as positive numbers). If the person staggers further he falls off the pier and well, stops walking. For our `while` loop, this implies that two conditions must be true for it to continue. They are evaluated from left to right – as soon as the first one fails, `R` stops testing and stops the loop. If the first condition passes, the second condition is checked. Depending on whether the condition is true or false, either the loop body is executed or the loop is stopped.

So let's see how far our drunk sailor gets...

```r
pos <- 0
step <- -1
while (pos >= -3 && pos <= 5) {
  cat("Step =", step <- step + 1, "\n")
  cat("Position =", pos, "\n")
  coin <- rbinom(n = 1, size = 1, prob = 0.5)
  if (coin == 1) { ## random walk
    pos <- pos + 1
  } else {
    pos <- pos - 1
  }
}
```

### 10.5 repeat

The `repeat` structure initiates an infinite loop. The only way to exit a `repeat` loop is to call `break`. In the following example, a typical setting of an iterative estimation process is simulated – it does not run, though. We start with the value `x0` and a tolerance limit of `tol`. Now a new estimate `x1` is estimated with the fictional function `computeEstimate()`. If the difference between `x0` and `x1` is greater than our tolerance limit, the new estimate is assigned to `x0` and the loop is rerun. This happens until the difference between the computed value `x1` and the reference `x0` falls below the tolerance `tol`. Then the loop is stopped.

```r
x0 <- 1
tol <- 1e-8

repeat {
  x1 <- computeEstimate(x0)
  if (abs(x1 - x0) < tol) {
    break
  } else {
    x0 <- x1
  }
}
```

This loop is a bit dangerous because there is no guarantee it will stop. It would be better to set a hard limit on the number of iterations (e.g., using a `for()` loop) and then report whether convergence was achieved or not.

### 10.6 next

In order to skip certain iterations of a loop, the `next` structure can be used.

```r
for (i in 1:100) {
  if (i <= 20) {
    ## Skip the first 20 iterations
    next
  }
  ## Do something here
}
```

## Further reading

- Wickham, H., & Grolemund, G. (2017). R for Data Science. Available under r4ds.had.co.nz/

## Exercises

**1.** For each of the following code sequences, predict the value of `answer` after running the loop. Then do the computation.

**a)**

```
answer <- 0
for (j in 3:5) {answer <- j + answer}
```

**b)**

```
answer <- 10
for (j in 3:5) {answer <- j + answer}
```

**c)**

```
answer <- 0
for (j in 3:5) {answer <- j * answer}
```

**d)**

```
answer <- 10
for (j in 3:5) {answer <- j * answer}
```

**e)**

```
answer <- NULL
for (j in 3:5) {answer <- c(answer, j)}
```

**f)**

```
answer <- c(2, 7, 1, 5, 12, 3, 4)
for (j in 2:length(answer)) {
  answer[j] <- max(answer[j], answer[j - 1])
}
```

**2.** The $\sqrt{N}$ law: The precision of the sample average improves with the square root of the sample size $N$. Simulate $10^6$ Poisson-distributed random numbers with expectation value $\lambda = 100$ via

```
set.seed(1)  # see ?set.seed for details
x <- rpois(n = 1e6, lambda = 100)
```

**a)** Write a loop (`repeat` or `while`) which calculates the mean, variance, standard deviation, and standard error of the first $N$ elements of `x` until `se` $\leq 0.05$. Start with $N = 2$ and multiply $N$ after each iteration by a factor of 2. Store $N$ and the calculated values for each iteration as rows in a matrix named `result`. Use the function `meanVarSdSe()` developed in Section 8.6. Make sure that the column names of your result matrix match the following output. Your matrix should look like

```
tail(result)  #  see ?tail for details
```

```
##            N      mean       var       sd          se
## [11,]  2048  99.90137 104.63414 10.22908 0.22603293
## [12,]  4096  99.97803 101.65813 10.08257 0.15754008
## [13,]  8192 100.02466  99.13747  9.95678 0.11000792
## [14,] 16384 100.00885 100.58669 10.02929 0.07835384
## [15,] 32768 100.04257 101.13827 10.05675 0.05555623
## [16,] 65536  99.97209 100.32065 10.01602 0.03912508
```

Are these values plausible?

**b)** Explain why a `for` loop is not optimal for this specific task.

**c)** Lower the break condition to `se` $\leq 0.01$. Run the changed code. Make sure that you reinitialize `N` and `result` before running the loop. What happens? Compare `N` and `length(x)`. Which expression causes the error?

**d)** The $\sqrt{N}$ law can formally be expressed as

$$se = c \cdot \frac{1}{\sqrt{N}} = cN^{-\frac{1}{2}} \tag{2}$$

$$\Leftrightarrow \log(se) = \log(c) + \log\left(N^{-\frac{1}{2}}\right) \tag{3}$$

$$\Leftrightarrow \log(se) = \log(c) - \frac{1}{2}\log(N), \tag{4}$$

with $c$ being a constant. Fit a linear model to eq. (4) via

```r
lmResult <- lm(log(se) ~ log(N), data = data.frame(result))
summary(lmResult)
```

```
##
## Call:
## lm(formula = log(se) ~ log(N), data = data.frame(result))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.17541 -0.05660  0.01691  0.02669  0.37975
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.26831    0.06804   33.34 9.71e-15 ***
## log(N)      -0.49841    0.01015  -49.10  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1297 on 14 degrees of freedom
## Multiple R-squared:  0.9942, Adjusted R-squared:  0.9938
## F-statistic:  2410 on 1 and 14 DF,  p-value: < 2.2e-16
```

Discuss the output. How can you interpret the estimated coefficients in a $\log$ - $\log$ regression?

**3.** Design a statement that checks several conditions of `x`: whether it is not numeric, whether it is greater than zero, smaller than zero or whether it equals zero. Depending on what is true, return a character string stating which condition applies. You can use the `print` function here.

**4.** Imagine you have the following two vectors: `a` – containing names of students, and `b` – containing their grades.

```
a <- c("Peter", "Paul", "Mary", "Max", "Moritz")
b <- c("Grade", 3, 2, 1, 1, 5)
```

Use a `for` loop in order to print the names in combination with their grades. Peter has grade 3, Paul has grade 2 etc. Prepare your function such that the number of students could change without you having to modify the function.

Note that this exercise could be solved using a `mapply()` function, as well. So if you already know how, you are welcome to do just that.

**5.** Write a function called `cormatrix` that takes a dataframe as argument, selects the numeric variables from the dataframe, and prints a correlation matrix for these numeric variables. The variable names should be preserved in the correlation matrix. Note that it is about chosing the numeric columns here. The correlations themselves can be computed via the `cor()` function.

**6.** Let's get back to the drunken sailor of the `while` loop above (see Section 10.4):

```
pos <- 0
step <- -1
while (pos >= -3 && pos <= 5) {
  cat("Step =", step <- step + 1, "\n")
  cat("Position =", pos, "\n")
  coin <- rbinom(n = 1, size = 1, prob = 0.5)
  if (coin == 1) { ## random walk
    pos <- pos + 1
  } else {
    pos <- pos - 1
  }
}
```

Now imagine the drunken sailor needs a break every five steps. Use the `next` structure to modify the loop accordingly. Note that the modulo operation `%%` could be useful.

## Solutions

**1.** This question is best solved if we mentally go through each iteration of the loop and keep in mind the result of the iteration. This is hard for two reasons: first, remembering with which value of `answer` we started. And second, understanding the rather unclear notation of the loop. So a nice practice would be to explain in advance what the loop is programmed for.

**2.** The $\sqrt{N}$ law is crucial for doing statistics. A lot of concepts work the way they do because they assume the $\sqrt{N}$ law.

```
set.seed(1)  # see ?set.seed for details
x <- rpois(n = 1e6, lambda = 100)
```

**a)** Set the break condition and the factor by which `N` is multiplied after each iteration of the loop, first. Note that we should be sure that the break condition will be reached sometime, otherwise the main memory gets crowded.

```
tol <- 0.05
factor <- 2
```

```
meanVarSdSe <- function(x) {
  c(mean = mean(x),
    var = var(x),
    sd = sd(x),
    se = sd(x) / sqrt(length(x)))
}

result <- NULL
n <- 2

repeat {
  result <- rbind(result, c(N = n, meanVarSdSe(x[1:n])))
  if (result[nrow(result), "se"] <= tol) break
  n <- n * factor
}

tail(result)
```

```
##             N      mean       var       sd         se
## [11,]  2048  99.90137 104.63414 10.22908 0.22603293
## [12,]  4096  99.97803 101.65813 10.08257 0.15754008
## [13,]  8192 100.02466  99.13747  9.95678 0.11000792
## [14,] 16384 100.00885 100.58669 10.02929 0.07835384
## [15,] 32768 100.04257 101.13827 10.05675 0.05555623
## [16,] 65536  99.97209 100.32065 10.01602 0.03912508
```

The values returned are plausible indeed. The more values are involved in the computation, the more precise the estimation of the mean gets. Also, since the number of observations is used in the denominator of the standard error, the standard error becomes smaller as the sample size increases. Hence, the law of large numbers works.

**b)** When using a `for` loop, we have to prespecify the amount of iterations the loop is meant to run. In the example above, we are not sure how many iterations will be necessary to calculate the mean up to a certain precision. Hence, a `for` loop would not be adequate.

**c)** If the break condition is lowered, the number of iterations needed for convergence increases. Then the number of observations needed for the estimation exceeds the length of our prespecified vector. Hence, calculation of the mean, variance, standard deviation, and standard error return a missing value `NA`. When the missing value is compared to the break condition, the loop stops and an error is returned:

```
Error in if (result[nrow(result), "se"] <= tol) break :
  missing value where TRUE/FALSE needed
```

**2. d)** The regression output shows how the standard error `se` depends on the number of observations `N` used for calculation. The negative sign of the coefficient implies that the standard error decreases with growing sample size. Coming to the magnitude of the coefficient: If both, regressor and regressant, are on a logarithmic scale, the effect of one on the other can be interpreted in percentages. Hence, if the number of observations increases by one percent, the standard error decreases by about 0.5 percent. This result confirms equation 4 in that the coefficient for $log(N)$ is $-0.5$. The logarithm of the constant is estimated to be $log(c) = 2.27$.

```
lmResult <- lm(log(se) ~ log(N), data = data.frame(result))
summary(lmResult)
```

```
##
## Call:
## lm(formula = log(se) ~ log(N), data = data.frame(result))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.17541 -0.05660  0.01691  0.02669  0.37975
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.26831    0.06804   33.34 9.71e-15 ***
## log(N)      -0.49841    0.01015  -49.10  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1297 on 14 degrees of freedom
## Multiple R-squared:  0.9942, Adjusted R-squared:  0.9938
## F-statistic:  2410 on 1 and 14 DF,  p-value: < 2.2e-16
```

**3.** Note that in the following statement the order of queries is not chosen at random. It makes absolutely sense to check whether or not `x` is a number before it is checked whether or not `x` is positive or negative etc.

```
if (!is.numeric(x)) {
  print("x is not a number")
} else if (x > 0) {
  print("x is a positive number")
} else if (x < 0) {
  print("x is a negative number")
} else if (x == 0) {
  print("x equals zero")
}
```

**4.** The hardest challenge here is that vector `a` starts with the names right away while the grades in vector `b` start at the second position. The rest of the `for` loop should be straight forward.

```
a <- c("Peter", "Paul", "Mary", "Max", "Moritz")
b <- c("Grade", 3, 2, 1, 1, 5)

for (i in seq_along(a)) {
  print(c(a[i], b[i+1]))
}
```

```
## [1] "Peter" "3"
## [1] "Paul" "2"
## [1] "Mary" "1"
## [1] "Max" "1"
## [1] "Moritz" "5"
```

If you used a `mapply()` structure, this is your solution:

```
mapply(paste, a, b[-1])
```

```
##       Peter        Paul        Mary         Max      Moritz
##   "Peter 3"    "Paul 2"    "Mary 1"     "Max 1" "Moritz 5"
```

As in most of the cases, the `apply()` structure takes less lines than a `for` loop.

**5.** The following function uses a `for` loop which loops over the columns of the given dataframe. It checks for each column if it is numeric. If so, the column is added to an extra dataframe and its name is written to a character vector. When all columns have been checked, the names are written to the extra dataframe and the correlations are calculated using the `cor()` function.

```
cormatrix <- function(x) {
  df <- NULL
  cnames <- NULL
  for (i in names(x)) {
    if (is.numeric(x[, i])) {
      df <- cbind(df, x[, i])
      cnames <- c(cnames, names(x[i]))
    }
  }
  colnames(df) <- cnames
  print(cor(df))
}
```

**6.** In order to give the drunken sailor a break every five steps, we can use the `next` structure. The challenge is to identify the time when the sailor has taken five steps. This can be achieved using the modulo operation. If the number of steps modulo five equals 0, we know that the sailor has taken five additional steps. We can then skip one step using `next`.

```r
pos <- 0
step <- -1
while (pos >= -3 && pos <= 5) {
  cat("Step =", step <- step + 1, "\n")
  cat("Position =", pos, "\n")
  if (step %% 5 == 0 & step != 0) next
  coin <- rbinom(n = 1, size = 1, prob = 0.5)
  if (coin == 1) { ## random walk
    pos <- pos + 1
  } else {
    pos <- pos - 1
  }
}
```

# 11 *apply() Functions

Writing `for` or `while` loops is useful when programming iterative processes, i.e., when calculations depend on the outcome of previous elements – but for other functionalities, it is not particularly efficient. Especially when working interactively on the command line, loops are not easy to implement. There are some functions which implement looping to make life easier: the *`apply` family. *`apply` functions are fast. And if you use special packages so that you can do parallel computations on multicore systems, they are super-fast. The most important functions can be found in the table below.

| **Function** | **Purpose** |
|---|---|
| `apply()` | apply a function to the rows or columns of a matrix |
| `lapply()` | apply a function to each element of a list and get a list back |
| `sapply()` | apply a function to each element of a vector or list and get a vector back |
| `vapply()` | safer and faster version of sapply() |
| `mapply()` | multivariate version of sapply() |
| `tapply()` | apply a function to (two or more) groups of values given by the levels of one (or more) factor(s) |
| `rapply()` | apply a function recursively to each element of a nested list |

## 11.1 apply()

`apply()` is used to evaluate a function over the margins of an array, e.g., summing up the rows or columns of a matrix. It is not neccessarily faster than writing a loop, but it works in one line!

```
x <- matrix(rnorm(20), nrow = 5, ncol = 4)
x
```

```
##             [,1]        [,2]       [,3]       [,4]
## [1,] -0.60058785 -0.25504236  0.1968360  0.0744612
## [2,] -1.68278438  0.02961438  1.0129349  0.8686395
## [3,]  0.03157764  1.53510934 -0.3002391  2.0420025
## [4,] -0.65549812 -1.10722485  0.6176942 -1.2371123
## [5,]  0.60756150  1.00688365  1.2452776 -0.8180228
```

```
apply(x, 2, mean)
```

```
## [1] -0.4599462  0.2418680  0.5545007  0.1859936
```

For sums and means of matrix dimensions, there are the following shortcuts:

```
rowSums   = apply(x, 1, sum)
rowMeans  = apply(x, 1, mean)
colSums   = apply(x, 2, sum)
colMeans  = apply(x, 2, mean)
```

The shortcut functions are *much* faster, but you won't notice unless you're using a large matrix.

### 11.2 lapply()

`lapply()` applies a function to each element of a list and takes two or more arguments: a list `X`, a function `FUN` (or the name of a function), and other arguments via its `...` argument. If `X` is not a list, it will be coerced to a list using `as.list()`. The actual looping is done internally in C code which makes it fast. `lapply()` always returns a list, regardless of the class of the input.

The following example demonstrates how one can take the mean over the elements `a` and `b` of a list.

```
set.seed(1)
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.1322028
```

In the next example, uniformly distributed variables are generated. The elements of the list `nElements` control how many numbers are generated each time the function `runif()` is executed.

```
nElements <- 1:4
lapply(nElements, runif)
```

```
## [[1]]
## [1] 0.9347052
##
## [[2]]
## [1] 0.2121425 0.6516738
##
## [[3]]
## [1] 0.1255551 0.2672207 0.3861141
##
## [[4]]
## [1] 0.01339033 0.38238796 0.86969085 0.34034900
```

## 11.3 sapply()

`sapply()` tries to simplify the result of `lapply()` if possible. A vector is returned if the result is a list where every element is of length one. A matrix is returned if the result is a list where every element is a vector of the same length (> 1). A list is returned if `R` does not know how to simplify. The `sapply()` argument `simplify` – responsible for simplification – is described in the manual.

Below we repeat the command just seen with `lapply()` using `sapply()` instead. What becomes clear is how `R` simplifies the result turning it from a list to a vector.

```
set.seed(1)
x <- list(a = 1:5, b = rnorm(10))
sapply(x, mean)
```

```
##         a         b
## 3.0000000 0.1322028
```

A somewhat safer version of `sapply()` is the function `vapply()`. It is safer because you can pre-specify the type of the return value so it shouldn't be able to produce surprising results. Sometimes it may even be faster than `sapply()` because the output type is already known and does not need to be determined inside the function.

## 11.4 mapply()

`mapply()` is the multivariate version of `sapply()`. It takes two or more vectors or lists and applies a function in parallel over their first elements, their second elements and so on. If the vectors or lists do not have the same length, the elements of the shorter ones are recycled.

In the following example we have five students Peter, Paul, Mary, Max and Moritz and their respective grades. We want to generate a vector containing the grade next to a student's name. So we call the `mapply()` function in combination with the `paste()` command.

```
students <- c("Peter", "Paul", "Mary", "Max", "Moritz")
grades <- c(3, 2, 1, 1, 5)
mapply(paste, students, grades, USE.NAMES = FALSE)
```

```
## [1] "Peter 3"  "Paul 2"   "Mary 1"   "Max 1"    "Moritz 5"
```

An example for a function with three arguments is the `rnorm` function. Here we generate a list of vectors containing random numbers drawn from normal distributions. The first vector contains three numbers from a normal distribution with mean four and standard deviation two. The second vector contains four numbers from a normal distribution with mean three and standard deviation two and so on. Note here that, since the argument for the standard deviation is a vector of length one, but we want to generate four vectors of random numbers, the two is recycled four times.

```
mapply(rnorm, 3:6, 4:1, 2)
```

```
## [[1]]
## [1] 7.023562 4.779686 2.757519
```

```
##
## [[2]]
## [1] -1.429400  5.249862  2.910133  2.967619
##
## [[3]]
## [1] 3.887672 3.642442 3.187803 3.837955 3.564273
##
## [[4]]
## [1]  1.1491300 -2.9787034  2.2396515  0.8877425  0.6884090 -1.9415048
```

### 11.5 tapply()

The `tapply()` function is used in order to apply a function to subsets of a vector, i.e., to (two or more) groups of values given by the levels of one (or more) factors. It is very flexible. `tapply()` requires three parameters: The first parameter specifies the metric variable that is examined. As second parameter, one factor (or a list of multiple factors) that group(s) the first variable is defined, and the last parameter is the name of the function that is to be applied. In the following example, we will work with a dataset called `analytics`, which we have to load into the workspace first. Then, using `tapply()`, the mean of the average time on page is calculated for the subsets of men and women:

```
library(readr)
analytics <- read_csv(file = "Data/Analytics.csv")
```

```
tapply(analytics$avgTimeOnPage, analytics$userGender, FUN = mean)
```

```
##   female     male
## 141.1393 112.5616
```

It is possible to use any other function. In the following example, the `length()` function, which counts the number of elements, is used in order to build a frequency table:

```
tapply(analytics$avgTimeOnPage, analytics$userGender, FUN = length)
```

```
## female   male
##     27     75
```

Additionally, it is fairly easy to group the first variable according to more than one factor. In order to do so, a list of factors is passed to the function:

```
tapply(analytics$avgTimeOnPage,
       list(analytics$bouncesHigh, analytics$userGender),
       FUN = mean)
```

```
##                female      male
## <10 bounces  118.1055  96.55451
## >=10 bounces 152.6563 126.56775
```

## 11.6 rapply()

The `rapply()` function works like the `lapply()` function on a list. But it goes further and applies the specified function recursively to all elements of nested list objects. The beauty of it is that you can specify on which classes of the elements the function should operate. Of course, it is important to have your classes correctly specified.

First, we start with our `analytics` dataset. The first operation is intended for character variables only. It counts the length of the strings:

```
rapply(analytics,
       nchar,
       classes = "character",
       how = "unlist")
```

Since the argument `how = "unlist"` is specified, only the respective result is returned. Next, we want to round the decimals of all numeric variables to 0 and rewrite them into the variables:

```
rapply(analytics,
       function(x) round(x, 0),
       classes = "numeric",
       how = "replace")
```

```
## # A tibble: 102 x 10
##        date users newUsers sessions bounces sessionDuration pageviews
##       <int> <int>    <int>    <int>   <int>           <int>     <int>
##  1 20140521    12       11       13      10             425        18
##  2 20140521    13       10       14      10             703        26
##  3 20140526    12       10       15      10            2596        36
##  4 20140527    14       11       21       8            5605        82
##  5 20140528    16       14       18      10            3529        52
##  6 20140530    12       10       12       7            2364        46
##  7 20140602    15       12       15      12              36        18
##  8 20140606    10        8       11       7             963        31
##  9 20140611    18       15       21      12            3005        39
## 10 20140617    16       15       16      13            1787        22
## # ... with 92 more rows, and 3 more variables: avgTimeOnPage <dbl>,
## #   userGender <chr>, bouncesHigh <chr>
```

In order to demonstrate `rapply()`'s power to apply functions recursively to all elements of a nested list, we use a slightly modified example taken from the help page of the function.

```
nestedList <- list(list(a = pi, b = list(c = 2)), d = "a test")
nestedList
```

```
## [[1]]
## [[1]]$a
## [1] 3.141593
##
## [[1]]$b
## [[1]]$b$c
## [1] 2
##
##
##
## $d
## [1] "a test"
```

The first function operates on numerics only, takes the square root and replaces the according values. The character elements are not touched and returned as before.

```
rapply(nestedList,
       sqrt,
       classes = "numeric",
       how = "replace")
```

```
## [[1]]
## [[1]]$a
## [1] 1.772454
##
## [[1]]$b
## [[1]]$b$c
## [1] 1.414214
##
##
##
## $d
## [1] "a test"
```

The next function takes the logarithm to the base of two of the numeric elements. So additionally to the standard arguments, the argument `base = 2` is passed through to the function of the logarithm.

```
rapply(nestedList,
       log,
       classes = "numeric",
       how = "replace",
       base = 2)
```

```
## [[1]]
## [[1]]$a
## [1] 1.651496
##
## [[1]]$b
## [[1]]$b$c
## [1] 1
##
##
##
## $d
## [1] "a test"
```

## 11.7 split

The auxiliary function `split()` can be especially useful in combination with `lapply()` or `sapply()`. It takes a vector or other objects and splits it into groups determined by a factor or list of factors. We continue working with the `analytics` dataset by picking the variables `sessionDuration`, `avgTimeOnPage`, and `bounces` and splitting the resulting dataset in blocks according to the `userGender`.

```
s <- split(analytics[, c("sessionDuration", "avgTimeOnPage", "bounces")],
           f = analytics$userGender)
```

Then we can compute the column means for each of the variables in the blocks using an `sapply()`:

```
sapply(s, colMeans, na.rm = TRUE)
```

```
##                     female      male
## sessionDuration 1622.96296 2871.6533
## avgTimeOnPage    141.13934  112.5616
## bounces           10.22222   10.3200
```

Another common idiom is to split-apply-and-combine data.

```
dataList <- lapply(split(analytics$avgTimeOnPage,
                         f = analytics$userGender,
                         drop = TRUE),
                   quantile, na.rm = TRUE)
do.call(cbind, dataList)
```

```
##         female      male
## 0%    16.58333  12.00000
## 25%   62.44348  59.39943
## 50%  144.75000  96.60000
## 75%  213.46528 148.91667
## 100% 325.66667 384.40000
```

This is equivalent to:

```
sapply(split(analytics$avgTimeOnPage,
             f = analytics$userGender,
             drop = TRUE),
       quantile, na.rm = TRUE)
```

```
##         female      male
## 0%    16.58333  12.00000
## 25%   62.44348  59.39943
## 50%  144.75000  96.60000
## 75%  213.46528 148.91667
## 100% 325.66667 384.40000
```

Although `sapply()` automates the "combine" step, it may not always do what you expect! So it is better to use the `do.call(FUN, list)` to control the output structure.

A similar example is the following: We want to split a character vector with names of very important people and recombine the results as a data frame. So after defining our character vector with the names, we use `strsplit()` in order to split the first name from the family name and write each person into one element of a list.

```
names <- c("Adam Riese", "Albert Einstein", "Oliver Kahn")
namesList <- strsplit(names, split = " ")
str (namesList)
```

```
## List of 3
##  $ : chr [1:2] "Adam" "Riese"
##  $ : chr [1:2] "Albert" "Einstein"
##  $ : chr [1:2] "Oliver" "Kahn"
```

Next, we put the first and last name of a person into the columns of a data frame – we do this for each person separately using `lapply()`. And finally, we recombine the one-row data frames to a single data frame.

```
namesData <- lapply(namesList,
                    function(x) data.frame(firstName = x[[1]],
                                           familyName = x[[2]]))
do.call(rbind, namesData)
```

```
##   firstName familyName
## 1      Adam      Riese
## 2    Albert   Einstein
## 3    Oliver       Kahn
```

## Further reading

- Wickham, H. Advanced R (2017). *(Chapter 11)* Available under adv-r.had.co.nz/Functionals.html

## Exercises

**1.** Let's look at the `apply()` function, used to apply a function over the rows or columns of a matrix or array. First, generate the matrix `matA` via

```
set.seed(1)
matA <- matrix(sample(24), ncol = 6)
matA
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    7    5   11   23    6   17
## [2,]    9   18    1   21   24   15
## [3,]   13   19    3    8   22    2
## [4,]   20   12   14   16    4   10
```

**a)** Sort the columns of the matrix in ascending order. Your result should look like

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    7    5    1    8    4    2
## [2,]    9   12    3   16    6   10
## [3,]   13   18   11   21   22   15
## [4,]   20   19   14   23   24   17
```

**b)** Sort the columns of the matrix in descending order. Your result should look like

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   20   19   14   23   24   17
## [2,]   13   18   11   21   22   15
## [3,]    9   12    3   16    6   10
## [4,]    7    5    1    8    4    2
```

**c)** Sort the rows of the matrix in descending order. Your result should look like

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   23   17   11    7    6    5
## [2,]   24   21   18   15    9    1
## [3,]   22   19   13    8    3    2
## [4,]   20   16   14   12   10    4
```

**2.** Let's work with the example for an `apply()` implementation from above (Section 11.1):

```
x <- matrix(rnorm(20), nrow = 5 , ncol = 4)
x
```

```
##             [,1]      [,2]        [,3]       [,4]
## [1,] -0.62124058 0.9438362  0.07456498 -1.4707524
## [2,] -2.21469989 0.8212212 -1.98935170 -0.4781501
## [3,]  1.12493092 0.5939013  0.61982575  0.4179416
## [4,] -0.04493361 0.9189774 -0.05612874  1.3586796
## [5,] -0.01619026 0.7821363 -0.15579551 -0.1027877
```

```
apply(x, 2, mean)
```

```
## [1] -0.35442668  0.81201448 -0.30137704 -0.05501381
```

Now take the same matrix and calculate the 25% and the 75% quantile of the rows instead of the mean over the columns (of course we have very few observations such that quantiles do not really make sense. But we want to hold the matrix small for demonstration purposes).

**3.** Next, we generate a 2 by 2 by 10 array of the following form:

```
a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
```

Using `rowMeans()`, it is possible to build means over the third dimension:

```
rowMeans(a, dims = 2)
```

```
##             [,1]      [,2]
## [1,]  0.08132523 0.2545651
## [2,] -0.00442013 0.3690307
```

Now go ahead and use the `apply()` function in order to obtain the same result.

**4.** Create a list of matrices via

```
set.seed(1)
n <- 5      # number of list elements that will be created
matNames <- paste("m", 1:n, sep = "_")
matList <- list()

for (i in matNames) {
  cols  <- sample(2:5, 1)
  rows  <- sample(2:5, 1)
  matList[[i]] <- matrix(runif(rows * cols),
                         ncol = cols, nrow = rows)
}

str(matList)
```

```
## List of 5
##  $ m_1: num [1:3, 1:3] 0.573 0.908 0.202 0.898 0.945 ...
##  $ m_2: num [1:4, 1:2] 0.384 0.77 0.498 0.718 0.992 ...
##  $ m_3: num [1:4, 1:2] 0.1256 0.2672 0.3861 0.0134 0.3824 ...
##  $ m_4: num [1:3, 1:4] 0.186 0.827 0.668 0.794 0.108 ...
##  $ m_5: num [1:2, 1:5] 0.477 0.732 0.693 0.478 0.861 ...
```

**a)** Calculate the dimensions with `lapply()` for each matrix contained in `matList`. The `sapply()` output should be equal to

```
##      m_1 m_2 m_3 m_4 m_5
## [1,]   3   4   4   3   2
## [2,]   3   2   2   4   5
```

**b)** Check for each list element if it is a quadratic matrix with `sapply()/lapply()` and the following function:

```
is.quadratic <- function(x) nrow(x) == ncol(x)
```

**c)** Repeat the exercise in b) above using an *anonymous function*.

**d)** Select the subset of `matList` that only contains quadratic matrices. The result should look like

```
## $m_1
##           [,1]      [,2]       [,3]
## [1,] 0.5728534 0.8983897 0.62911404
## [2,] 0.9082078 0.9446753 0.06178627
## [3,] 0.2016819 0.6607978 0.20597457
```

**5.** Let's use the `lapply()` function in order to see if we can discover the law of large numbers. To this end, you can create a list of vectors of different length filled with normally distributed random numbers (`rnorm()`). Then, calculate the mean for each vector. According to the law of large numbers, in general, the mean of the longest vector should come closest to the theoretical mean of your random numbers.

**6.** Remember the `for` loop above where we built the dynamic vector of results for our capital stock of the next five years (see page 180).

```
capital <- 100
n <- 5
for (i in 1:n) {
  capital <- c(capital, capital[1] + i * 50)
}
```

Now rebuild the function using a function from the `*apply` family. Take both versions, rerun them making predictions for the next 10000 years, and look at the system time used to execute the commands. What do you notice?

**7.** Now we create another list of matrices using `mapply()`. First, create the following objects:

```
set.seed(1)
n <- 5
ncolVector <- sample(2:5, n, replace = TRUE)
nrowVector <- sample(2:5, n, replace =TRUE)
dataList <- lapply(ncolVector * nrowVector, runif)
```

**a)** Make sure you understand what `lapply(ncolVector * nrowVector, runif)` does.

**b)** Use the functions `mapply()`, `matrix()` and the objects `ncolVector`, `nrowVector`, and `dataList` to create a list of matrices. The numbers of `dataList` should thereby be filled into matrices with the respective number of columns and number of rows contained in the vectors `ncolVector` and `nrowVector`. Your result should look like:

```
str(matList)
```

```
## List of 5
##  $ : num [1:5, 1:3] 0.206 0.177 0.687 0.384 0.77 ...
##  $ : num [1:5, 1:3] 0.3861 0.0134 0.3824 0.8697 0.3403 ...
##  $ : num [1:4, 1:4] 0.821 0.647 0.783 0.553 0.53 ...
##  $ : num [1:4, 1:5] 0.316 0.519 0.662 0.407 0.913 ...
##  $ : num [1:2, 1:2] 0.864 0.39 0.777 0.961
```

**c)** Expand the `mapply(matrix, ...)` expression of part b) above so that the matrices are filled by rows. The result should be:

```
str(matList)
```

```
## List of 5
##  $ : num [1:5, 1:3] 0.206 0.384 0.718 0.777 0.652 ...
##  $ : num [1:5, 1:3] 0.386 0.87 0.6 0.827 0.108 ...
##  $ : num [1:4, 1:4] 0.821 0.53 0.732 0.438 0.647 ...
##  $ : num [1:4, 1:5] 0.316 0.294 0.479 0.839 0.519 ...
##  $ : num [1:2, 1:2] 0.864 0.777 0.39 0.961
```

**8.** Look at the following list of vectors:

```
lHard <- list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

Why do you think we called it `lHard`? Well, of course there is an easier way to produce the same output. So go ahead and rewrite the expression using `mapply`.

**9.** Load the `airquality` dataset via

```
data(airquality)    # the 'data' function loads the dataset
summary(airquality) # ?summary for details
```

```
##      Ozone           Solar.R           Wind             Temp      
##  Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00  
##  1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00  
##  Median : 31.50   Median :205.0   Median : 9.700   Median :79.00  
##  Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88  
##  3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00  
##  Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00  
##  NA's   :37       NA's   :7                                       
##      Month            Day      
##  Min.   :5.000   Min.   : 1.0  
##  1st Qu.:6.000   1st Qu.: 8.0  
##  Median :7.000   Median :16.0  
##  Mean   :6.993   Mean   :15.8  
##  3rd Qu.:8.000   3rd Qu.:23.0  
##  Max.   :9.000   Max.   :31.0  
## 
```

**a)** Calculate the mean temperature for each month separately with

```
tapply(airquality$Temp,
       INDEX = airquality$Month,
       mean,
       na.rm = TRUE)
```

```
##        5        6        7        8        9 
## 65.54839 79.10000 83.90323 83.96774 76.90000
```

**b)** Now calculate the mean `Ozone` concentration for each month. Your result should be

```
##        5        6        7        8        9
## 23.61538 29.44444 59.11538 59.96154 31.44828
```

**c)** Which parameter do you have to change to obtain the following output?

```
## $`5`
## [1] 23.61538
##
## $`6`
## [1] 29.44444
##
## $`7`
## [1] 59.11538
##
## $`8`
## [1] 59.96154
##
## $`9`
## [1] 31.44828
```

**10.** We have seen how to find the means of the average time on page for men and women above (see page 194). Now go ahead and calculate the ranges of the average time on page within these groups.

**Solutions**

**1.** Let's turn to the solution for the `apply()` exercise now. To start with, we do just simple sorting on the rows or columns of a matrix.

**a)** You can sort the columns of the matrix using the following command. The `2` stands for the columns, the command `sort` uses ascending order by default.

```
set.seed(1)
matA <- matrix(sample(24), ncol = 6)
matA
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    7    5   11   23    6   17
## [2,]    9   18    1   21   24   15
## [3,]   13   19    3    8   22    2
## [4,]   20   12   14   16    4   10
```

```
apply(matA, 2, sort)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    7    5    1    8    4    2
## [2,]    9   12    3   16    6   10
## [3,]   13   18   11   21   22   15
## [4,]   20   19   14   23   24   17
```

**b)** In order to reverse the sorting direction, we can simply add the parameter `decreasing = TRUE` to our previous statement.

```
apply(matA, 2, sort, decreasing=TRUE)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   20   19   14   23   24   17
## [2,]   13   18   11   21   22   15
## [3,]    9   12    3   16    6   10
## [4,]    7    5    1    8    4    2
```

**c)** Now, sorting the rows seems to be a little bit tricky. If we simply exchange the argument `2` for a `1`, the right operation is done but additionally, the matrix is transposed. Hence, in order to get back the right format, we have to transpose the result.

```
t(apply(matA, 1, sort, decreasing=TRUE))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   23   17   11    7    6    5
## [2,]   24   21   18   15    9    1
## [3,]   22   19   13    8    3    2
## [4,]   20   16   14   12   10    4
```

**2.** Luckily, the `apply()` function has the nice feature that via `...`, additional arguments can be passed through. So we simply chose the `quantile()` function as function and add a vector with the required quantiles as additional argument.

```
x <- matrix(rnorm(20), 5, 4)
apply(x, 1, quantile, probs = c(0.25, 0.75))
```

```
##           [,1]       [,2]      [,3]        [,4]       [,5]
## 25% -0.8336185 -2.0456887 0.5499114 -0.04773239 -0.1160397
## 75%  0.2918828 -0.1533072 0.7461020  1.02890292  0.1833914
```

**3.** Working with `apply()` in a 2-dimensional space is rather straightforward. It gets a little bit more difficult if the number of dimensions increases. In the following case, we have a total of three dimensions that computations could be done over. Reading the help page of `apply()` in detail helps to chose the right indices.

```
a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
apply(a, c(1, 2), mean)
```

```
##             [,1]      [,2]
## [1,]  0.08132523 0.2545651
## [2,] -0.00442013 0.3690307
```

**4.** After we have followed the instructions, we gained a rather complex construct of matrices called `matList`. It is a list containing differently sized matrices. In order to work with them, the functions `lapply()` and `sapply()` can be used.

**a)** For calculating the dimensions of the matrices we use the `dim()` function. As expected, the output of the `sapply()` is somewhat less complex than `lapply()`'s output.

```
str(matList)
```

```
## List of 5
##  $ m_1: num [1:3, 1:3] 0.573 0.908 0.202 0.898 0.945 ...
##  $ m_2: num [1:4, 1:2] 0.384 0.77 0.498 0.718 0.992 ...
##  $ m_3: num [1:4, 1:2] 0.1256 0.2672 0.3861 0.0134 0.3824 ...
##  $ m_4: num [1:3, 1:4] 0.186 0.827 0.668 0.794 0.108 ...
##  $ m_5: num [1:2, 1:5] 0.477 0.732 0.693 0.478 0.861 ...
```

```

sapply(matList, dim)
```

```
##      m_1 m_2 m_3 m_4 m_5
## [1,]   3   4   4   3   2
## [2,]   3   2   2   4   5
```

```
lapply(matList, dim)
```

```
## $m_1
## [1] 3 3
##
## $m_2
## [1] 4 2
##
## $m_3
## [1] 4 2
##
## $m_4
## [1] 3 4
##
## $m_5
## [1] 2 5
```

**b)** Along the lines of the previous task, we can simply replace the `dim()` function with our self-made function `is.quadratic()`.

```r
is.quadratic <- function(x) nrow(x) == ncol(x)

sapply(matList, is.quadratic)
```

```
##   m_1   m_2   m_3   m_4   m_5
##  TRUE FALSE FALSE FALSE FALSE
```

```r
lapply(matList, is.quadratic)
```

```
## $m_1
## [1] TRUE
##
## $m_2
## [1] FALSE
##
## $m_3
## [1] FALSE
##
## $m_4
## [1] FALSE
##
## $m_5
## [1] FALSE
```

**c)** An anonymous function is an unnamed function, like the right part of the `is.quadratic()` function defined above. Of course, in this explicit case it does not really make a difference whether we plug into the `sapply()` and `lapply()` the name of the function or its definition. Using anonymous functions can, however, be very powerful when it comes to executing several commands to the list elements. So it makes in each case sense to practice. . .

```r
sapply(matList, function(x) nrow(x) == ncol(x))
```

```
##   m_1   m_2   m_3   m_4   m_5
##  TRUE FALSE FALSE FALSE FALSE
```

```r
lapply(matList, function(x) nrow(x) == ncol(x))
```

```
## $m_1
## [1] TRUE
```

```
##
## $m_2
## [1] FALSE
##
## $m_3
## [1] FALSE
##
## $m_4
## [1] FALSE
##
## $m_5
## [1] FALSE
```

**d)** The output of the previous task is a list including `TRUE`s where a matrix is quadratic and `FALSE`s where a matrix is not quadratic In order to chose the quadratic matrices now, we can simply use this output as subsetting statement for the original matrix list.

```
matList[sapply(matList, function(x) nrow(x) == ncol(x))]
```

```
## $m_1
##           [,1]      [,2]       [,3]
## [1,] 0.5728534 0.8983897 0.62911404
## [2,] 0.9082078 0.9446753 0.06178627
## [3,] 0.2016819 0.6607978 0.20597457
```

**5.** Our first task is it to generate a list of vectors filled with normally distributed numbers. For simplicity reasons, we take the same mean for each vector of, e.g., 5, but vary the length of the vectors. If we now use the `lapply()` function for calculating the mean, we find the law of large numbers confirmed. The longer the vector, the closer the mean comes to 5. Note that in magnitudes of 10 or 50 entries, the law of large numbers is not observable yet.

```
x <- list(a = rnorm(10, 5),
          b = rnorm(50, 5),
          c = rnorm(100, 5),
          d = rnorm(500, 5))

lapply(x, mean)
```

```
## $a
## [1] 5.186496
##
## $b
## [1] 4.92949
##
## $c
## [1] 4.953681
##
## $d
## [1] 4.986891
```

**6.** This is a very nice exercise to demonstrate differences in performance of a `for` loop compared to, let's say, a `sapply()`. Already when doing simple computation in 10000 steps, the elapsed time with the `for` loop exceeds the time needed for the `sapply()`.

```
# for loop
res <- 100
system.time(
  for (i in 1:10000) {
    res <- c(res, res[1] + i * 50)
  }
)
```

```
##    user  system elapsed
##    0.25    0.00    0.25
```

```
# sapply
start <- 100
i <- 1:10000
system.time(res <- c(start, sapply(i, function(x) start + x * 50)))
```

```
##    user  system elapsed
##       0       0       0
```

**7.** You might have noticed that in the majority of exercises we start with setting a seed before we begin calculation. In order to ensure reproducibility of results, this is indispensible. Ok, so let's look at the solutions.

**a)** The `lapply()` constructs a list of vectors filled with uniformly distributed random numbers. The length of the vectors varies randomly and is determined by the two sampled vectors `ncolVector` and `nrowVector`.

**b)** `mapply()` offers the possibility to apply a function over the elements of different objects. In this case, we can use it to create the list of matrices. The function we apply is `matrix()`. Then the data contained in `dataList` is put into matrices while the vector `ncolVector` determines the number of columns and the vector `nrowVector` the number of rows.

```
matList <- mapply(matrix,
                  data = dataList,
                  ncol = ncolVector,
                  nrow = nrowVector)
```

**c)** In order to fill the matrices by rows, we need a way to pass through the argument `byrow = TRUE`. This is done as follows. Note that the one element in the list `MoreArgs` is recycled as often as needed.

```
matList <- mapply(matrix,
                  data = dataList,
                  ncol = ncolVector,
                  nrow = nrowVector,
                  MoreArgs = list(byrow = TRUE))
```

**8.** Using `mapply()` we are able to reduce the rather lengthy expression above to a very simple statement:

```r
## the smart way
lSmart <- mapply(rep, 1:4, 4:1)
lSmart
```

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

**9.** We are looking at the `airquality` dataset which contains variables about measures of airquality.

**a)** In order to calculate the mean temperature for each month, we can use the `tapply()` function.

```r
tapply(airquality$Temp,
       INDEX = airquality$Month,
       mean,
       na.rm = TRUE)
```

```
##        5        6        7        8        9
## 65.54839 79.10000 83.90323 83.96774 76.90000
```

**b)** Calculating the mean ozone concentration for each month, of course, is not very different. All you need to do is exchange the variable `Temp` with the variable `Ozone`.

```r
tapply(airquality$Ozone,
       INDEX = airquality$Month,
       mean,
       na.rm = TRUE)
```

```
##        5        6        7        8        9
## 23.61538 29.44444 59.11538 59.96154 31.44828
```

**c)** Internally, the `tapply()` function works on a list. Since the result so far is a named vector, some simplification must have taken place. And in fact, by adding the argument `simplify = FALSE`, a list is returned.

```r
tapply(airquality$Ozone,
       INDEX = airquality$Month,
       mean,
       na.rm = TRUE,
       simplify = FALSE)
```

```
## $`5`
## [1] 23.61538
##
## $`6`
## [1] 29.44444
##
## $`7`
## [1] 59.11538
##
## $`8`
## [1] 59.96154
##
## $`9`
## [1] 31.44828
```

**10.** Since within `R` there exists a `range()` function, it is straight forward to switch from calculating the mean for men and female to calculating the range. Just replace the function `mean()` with `range()`.

```
tapply(analytics$avgTimeOnPage,
       analytics$userGender,
       FUN = range)
```

```
## $female
## [1]  16.58333 325.66667
##
## $male
## [1]  12.0 384.4
```