Statistical Programming with R

Chapter 1 - Some history and getting started

Florian Ertz, Jan Pablo Burgard, and Ralf Münnich

Economic and Social Statistics
Economics / Faculty IV
Trier University

Winter semester 2021/2022

Copyright statement

A note on this course

It should be obvious that we cannot cover R in its entirety.

Our aim is to provide you with a solid foundation that enables you to follow our courses in the *M.Sc. Applied Statistics* and *M.Sc. Data Science* programmes on the one hand and to learn more about the language yourself (according to your specific needs) as you progress. Accordingly, we do not discuss each and every option of the presented functionality.

We aim for a *natural flow* in the exposition of contents. This means that we do not strictly move from one topic to another topic. Although we might seem to *jump* from time to time, there is a route we are taking.

Timeline

- **R**obert Gentleman and **R**oss Ihaka (both University of Auckland) start development of environment for statistical computing and graphics called *R* in 1992
- New language heavily influenced by *S* and *Scheme*
- First binary versions published in 1993
- First distribution as open-source software in 1995 (*GNU General Public License*)
- Formation of *R Development Core Team* in 1997
- Release of R 1.0.0 on 29 February 2000

References:
Ihaka, R. and R. Gentleman (1996): R: A Language for Data Analysis and Graphics. Journal of Computational and Graphical Statistics, 5(3), pp. 299–314.
Smith, D. (2020): The History of R (updated for 2020). URL: Link (16/09/2020).

What is R?

- R is a (statistical) programming language.
- R is a collection of statistical tools.
- R is a collection of graphical tools.
- R is not a program.

Core competencies

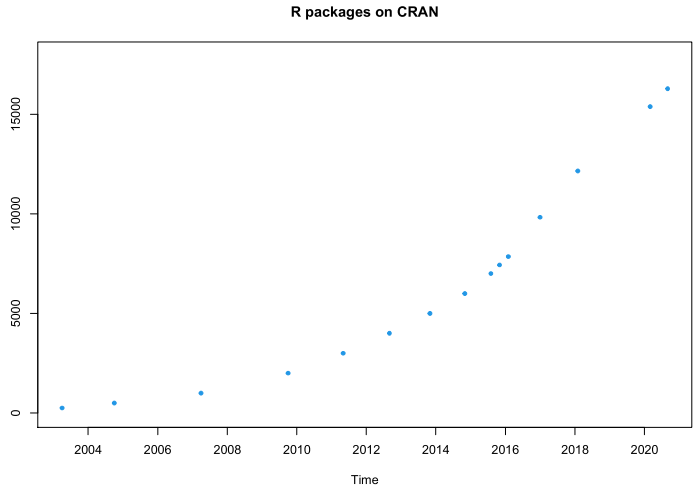
- R facilitates development and implementation of new methods (interpreted language)
- R incorporates most statistical methods
- R is *lingua franca* of statistics
- R facilitates data handling and manipulation
- R offers great visualisation options

The power of open-source software

- R does not cost money (*free as in beer*).
  → Large user base (2 million users in 2009 according to one estimate)
- R is open-source software (*free as in speech*).
  → Active user base and steady development
- R is platform-independent and therefore available for Microsoft Windows, Apple macOS, and Linux distributions.
- R can communicate with other tools like LaTeX, *Stata*, *SPSS*, etc.
  → *Reproducible research*
- R allows integration of other languages like *C/C++*, *Python*, etc.

Reference:
Vance, A. (2009): R You Ready for R?. URL: Link (16/09/2020).

R packages



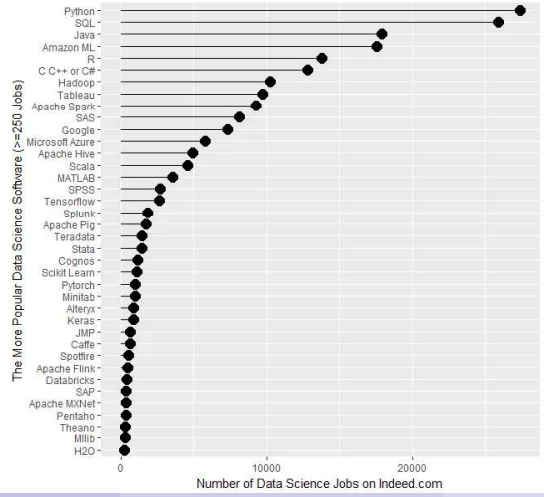
Data sources:
R-devel - R development and technical/programmer topics (2014).
https://cran.uni-muenster.de (different dates).

Drawbacks

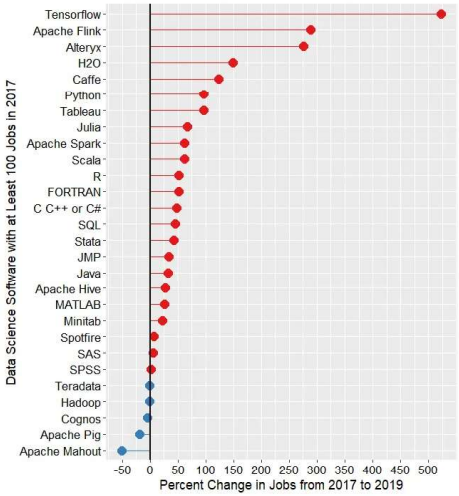
- R is not as fast as compiled languages for procedural tasks.
- R is *memory hungry*.
- R uses a command line interface (CLI).
  → Learning curve
  → Comprehension of applied methods
  → Reproducibility

Job market I

The following two plots are taken from Robert A. Muenchen's blog post *The Popularity of Data Science Software* which you can find here (16/09/2020).



Job market II



UseRs

You can reach a curated list of examples of corporate uses of R at Colin Fay's GitHub here (16/09/2020). Some *useRs* are:

Airbnb	Amazon Web Services	BBC
Booking	Dyson	eBay
The Economist	facebook	Financial Times
Google	John Deere	Microsoft
Netflix	New York Times	Süddeutsche Zeitung
T-Mobile	Twitter	Uber
UK Government	Wikimedia	Worldbank

R and visualisation

Paul Butler, a former facebook intern, demonstrated the power of R's visualisation tools with the plot below. For details, please see his blog (16/09/2020).

R base installation

The installation instructions in this subsection are based on websites accessed in September 2020. At a later point in time there might be small variations.

If you are working on a desktop in one of Trier University's PC pools, R is pre-installed and can be reached via the start menu. To install R with its default packages (more on extensions later) on your own computer, follow these steps:

1. Go to the project website https://www.r-project.org.
2. Follow the link **CRAN** (on the left under *Download*).
3. Select a *mirror* that is geographically close and follow its URL link. An example would be https://cran.uni-muenster.de under *Germany*.
4. Follow the download link for your operating system under *Download and Install R*.
5. Follow the link to the current version.
6. Follow installation procedure for your operating system.

Different ways to use R

After the base installation of R, depending on your operating system, you have the following options if you want to use R:

1. Pure CLI use in a terminal (Unix-based operating systems)
2. Graphical user interface (GUI) (Windows and macOS)

After the installation of additional tools, you have more options:

3. Text editor with plug-in (e.g. *Emacs*, *Vim*)
4. Integrated development environment (IDE) (e.g. *RStudio*)

RStudio offers a common interface across operating systems. We therefore use it in this course.

RStudio installation

If you are working on a desktop in one of Trier University's PC pools, RStudio is pre-installed and can be reached via the start menu. To install RStudio on your own computer, follow these steps:

1. Make sure that you already installed R itself.
2. Go to the website https://rstudio.com.
3. Hover over the *Products* tab and follow the link **RStudio**.
4. Follow the link **RStudio Desktop**.
5. Follow the link **DOWNLOAD RSTUDIO DESKTOP** under *Open Source Edition*.
6. Follow the link **DOWNLOAD** under *RStudio Desktop Open Source License*.
7. You can then usually click the blue button to download the recommended installer for your operating sytem.
8. Follow installation procedure for your operating system.

Updating existing installations

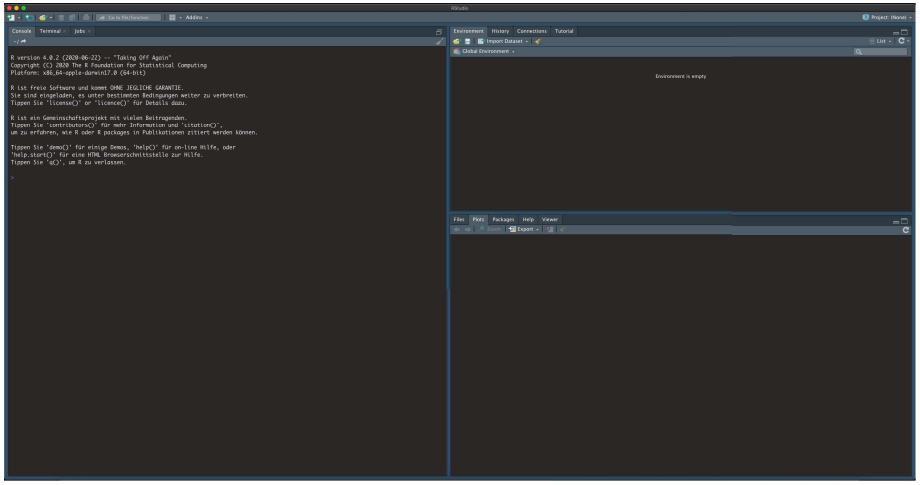
We would like to encourage you to update any already installed versions of R and RStudio to the newest versions in order to ensure compatibility with our e-learning and e-assessment system.

Some useful links

Should you encounter a problem along the way, the following links are good starting points to get help.

- https://www.r-project.org
- https://rstudio.com
- https://www.statmethods.net
- https://www.youtube.com/playlist?list=PLcgz5kNZFCkzSyBG3H-rUaPHoBXgijHfC
- https://stackoverflow.com/questions/tagged/r
- https://www.r-bloggers.com

A first look



Note that the dark colour scheme shown above is not the default, which is a bright scheme with a white background.

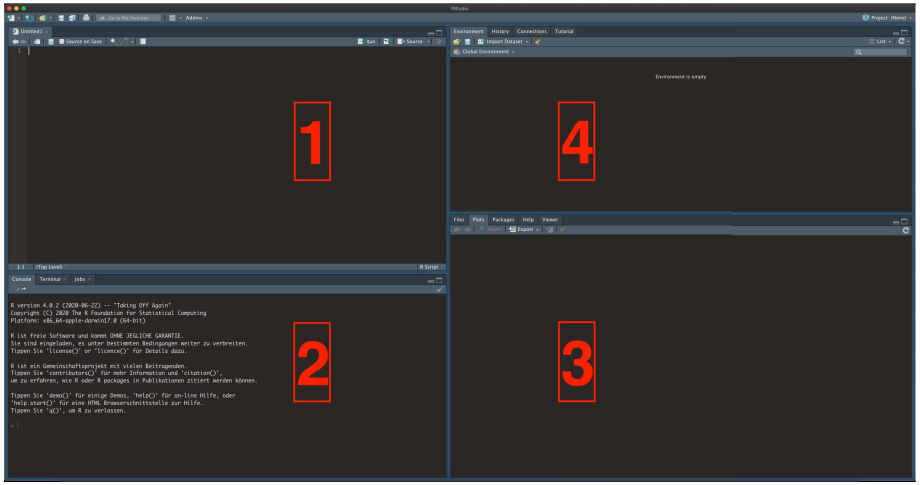
R scripts

Right when you start an R session, you should open an R script. To do so, click the button **New File** in the upper left-hand corner, indicated by a white sheet of paper with a green circle.

Then click the first item, which is **R Script**, indicated by a white sheet of paper with a blue circle.

ATTENTION:
If you open a simple text file by mistake, you cannot send code to the console later on.

Pane layout I



Pane layout II

The default pane layout (to which we will refer throughout this course) is as follows, starting in the upper left-hand corner and moving counter-clockwise (see red numbers on last slide):

1. Source (here: R script)
   Editor with syntax highlighting, bracket matching, and tooltips
2. Console
   Code execution and output (results, warning or error messages, etc.)
3. Plots, Help, etc.
4. Environment, History, etc.
   Overview of generated objects, history of commands, etc.

ATTENTION:
If some panes *disappear*, you can show all panes again by checking the last of the following entries: **View** → **Panes** → **Show All Panes**.

Programming in RStudio

A typical programming workflow in RStudio looks like this, where your activities are red, R's reactions are blue, and bracketed bullet points do not always happen:

1. Coding in pane 1
2. Checking of code in pane 1
3. Sending of (chunks of) code from pane 1 to pane 2
4. Reproduction of executed code in pane 2
5. (Output of warning or error messages in pane 2)
6. (Output of results in pane 2)
7. (Plotting or output of documentation in pane 3)
8. (Reading of warning or error messages in pane 2)
9. (Modification of code in pane 1 → Go to 2.)
10. (Use current results → Go to 1.)

Sourcing versus interactive mode

The workflow illustrated on the last slide is called sourcing and should be the one which you accustom yourself to. You can save your script for later use using the menu: **File** → **Save As...**. The resulting text file has the file extension **.R**. RStudio reminds you if the current state of your script has not been saved by changing the colour of the script name on its tab and adding an asterisk to it (for a script that has never been saved: *Untitled1**).

However, for very minor tasks like a small calculation that you only need once, you could also go straight to the console (pane 2) without using a script (interactive mode).

ATTENTION:
As soon as you work on something more complex and that you want to re-use, make sure to use sourcing.

Sending code to the console

There are different ways to send (previously selected chunks of) your code from the script to the console:

1. Hit the **Control** and **Enter** keys
2. Click button **Run** (upper right-hand corner of pane 1)
3. Click button **Source** (upper right-hand corner of pane 1) to send the **complete** script

If you are in interactive mode, however, you just type your code into the console and hit the **Enter** key to execute it.

ATTENTION:
By default, RStudio executes all lines of a multi-line command. To change this behaviour to a line-by-line execution (which we use here), use the menu: **Tools** → **Global Options...** → **Code** → **Editing** → **Ctrl + Enter executes:** → **Current line**.

Your first R program

Now you are ready to write and execute your first R program.

Throughout this course, your written code is highlighted by a background colour and R's output is indicated by two leading #s.

```
print("Hello world!")
```

```
## [1] "Hello world!"
```

Now in the console, the prompt (>) followed by the blinking cursor indicates that you can submit new commands.

Here, we used a function named `print`. Many function names in R are rather self-explanatory.

In R, a function name is always followed by parentheses which contain the function's arguments. These are used to define the user input and in turn specify how the function is executed to give a certain output.

Arithmetic I

R can handle arithmetic operations easily. Note that one command can extend to multiple lines (see the leading + in the console).

```
10 +
  5
## [1] 15
10 - 5
## [1] 5
10 * 5
## [1] 50
10 / 5
## [1] 2
15 %/% 7
## [1] 2
15 %% 7
## [1] 1
```

Arithmetic II

ATTENTION:
In R, the decimal separator is always a point.

Notice that we can easily nest functions like in the logarithm example below. R first evaluates the innermost function call and then progresses to the outer function(s).

```
sqrt(15)
## [1] 3.872983
10^5
## [1] 1e+05
exp(1)
## [1] 2.718282
log(exp(15))
## [1] 15
abs(-15)
## [1] 15
```


Arithmetic III

In the second line of code below (counting only input lines), we use three arguments in the function `sum`, where the **arguments are separated by commas**. Binary operators like + actually are functions too.

```
10 + 5 + 5
## [1] 20
sum(10, 5, 5)
## [1] 20
10 * 9 + 1 # Order of operations
## [1] 91
10 * (9 + 1) # Use of parentheses
## [1] 100
1 * 2 * 3 * 4 * 5 * 6
## [1] 720
factorial(6)
## [1] 720
```


Commenting scripts in R

You might have recognised the (additional) #s in the third and fourth line of code on the previous slide. These are the comment symbol in R.

All code that follows after the # in a line is not evaluated but just printed to the console.

You can use such comments to explain what your code actually does. This is not only useful for collaborators but also for yourself when you have to re-use some old code. However, keep your comments short and meaningful.

Comments can also be used to structure your script.

If you need your comment to stretch across multiple lines, start each line with a #.

Using at least five # in a row, you can collapse code chunks up to the next instance of five # in a row in RStudio. Check the small downward-facing arrow next to the line number in your script (pane 1).

Statistical Programming with R

Chapter 2 - Vectors, matrices, and arrays

Florian Ertz, Jan Pablo Burgard, and Ralf Münnich

Economic and Social Statistics
Economics / Faculty IV
Trier University

Winter semester 2021/2022

Copyright statement

The working directory

The default behaviour of R is to save scripts, data, plots etc. to the working directory. The default working directory will usually be your *Documents* folder.

You can very easily retrieve your current working directory.

```
getwd()
```

```
## [1] "/Users/florian/Seafile/StatProgR/Chapter_02"
```

Changing the working directory to another (already existing) folder is easy too. We will discuss more advanced options later.

```
setwd("../New_wd") # relative to current working directory
getwd()
## [1] "/Users/florian/Seafile/StatProgR/New_wd"
setwd("../Chapter_02") # back to the original working directo
```


Export of R output to a file

We can divert R console output to a text file (here in the working directory).

```
sink("Test.txt")
5 + 5
10 - 9
sink()
```

ATTENTION:
Note that the output (i.e. results) was really diverted and not printed to the console.

Scripts and working directories

For this course, we would encourage you to just use one central folder right from the start to avoid some headache caused by files located across multiple folders.

In a serious R workflow you will usually work with different folders for input data, output data, plots, etc. We will give you a glimpse of this later.

A handy trick is to just navigate to a saved script by file browser and open it. The script's folder will then automatically be taken to be the working directory. So try to avoid starting RStudio first and loading a script afterwards.

Umlauts and non-Latin letters

ATTENTION:
You should strictly avoid umlauts (ä, ö, etc.) and non-Latin (e.g. cyrillic) letters in script/file names, comments, and object names. Depending on the file encoding these can lead to some serious trouble and time-consuming bug hunting.

(Atomic) vectors

(Atomic) vectors are the most basic data structure in R.

Vectors in R are always column vectors.

We start with the discussion of numeric vectors and will get to know other types of vectors later.

To generate a vector containing the natural numbers from 1 through 10, we have the following options.

```
c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
##  [1]  1  2  3  4  5  6  7  8  9 10
seq(from = 1, to = 10, by = 1)
##  [1]  1  2  3  4  5  6  7  8  9 10
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
```


Function arguments – a closer look I

Let us look at the second command on the last slide again.

In the call of the function `seq` we used three arguments.

The names of these arguments were `from`, `to`, and `by`, respectively.

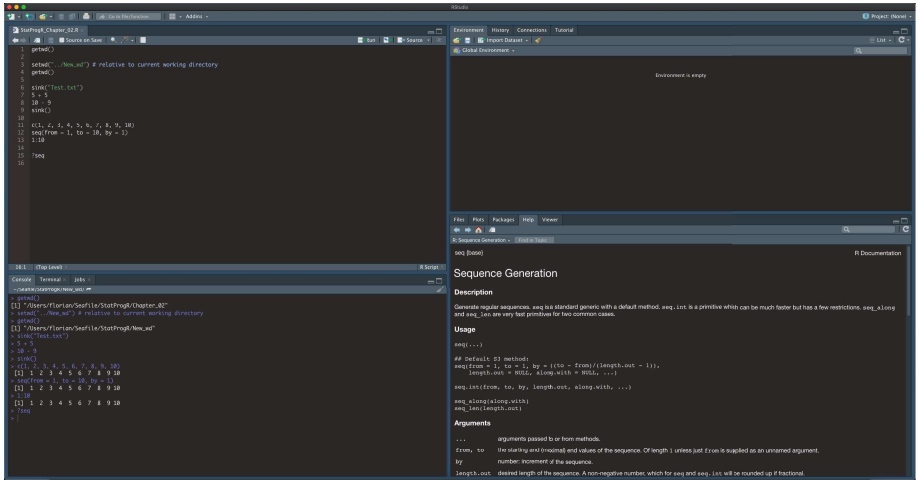
The values of these arguments were 1, 10, and 1, respectively.

R's help system I

Many times, the argument names are self-explanatory. Sometimes, though, we might need some guidance. R's built-in help system should be called in such cases.

```
?seq
```


R's help system II



R's help system III

R's help system (see pane 3) provides detailed information on functions:

1. **Description**
   Short description of what function does
2. **Usage**
   Information on argument order and default argument values of function
3. **Arguments**
   Most commonly used arguments of function
4. **Details**
   Details on usage of function, methods used, etc.
5. **Value**
   Output object of function
6. **References**
7. **See Also**
   Links to related functions
8. **Examples**

R's help system IV

If you do not want information on a specific function, you can do a more general search as well.

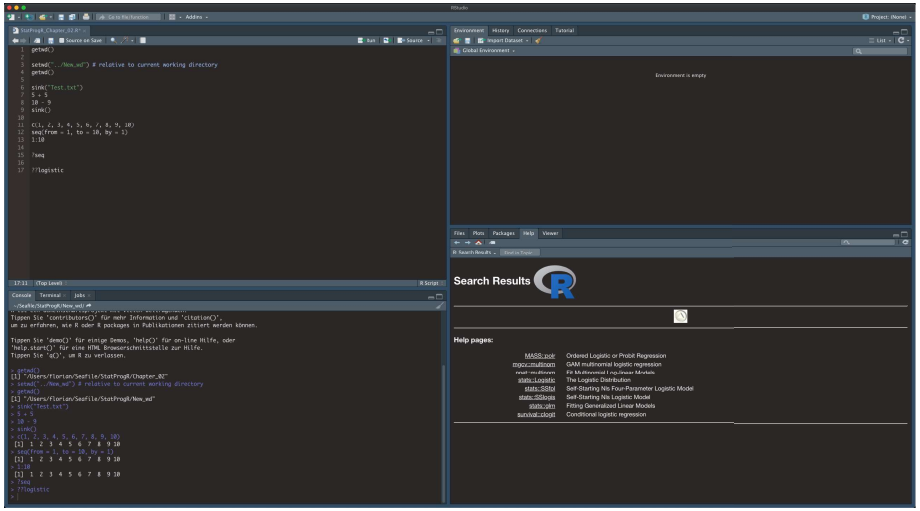
```
??logistic
```

To find functions that contain a certain character string in their names, we can use the function `apropos`.

```
apropos("sink")
```

```
## [1] "sink"          "sink.number"
```

R's help system V



Manuals and other material

You might want to check out the links shown in pane 3 after the following command `help.start()`.

Function arguments – a closer look II

We now understand that, for function `seq`, `from` defines the starting value of the resulting sequence, while `to` defines its end value. `by` sets the increment of the sequence.

If we **keep the argument order** of the *Usage* category of a function's help document in our function call, we can save ourselves some typing by **avoiding the argument names**.

```
seq(1, 10, 1)
```

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


Function arguments – a closer look III

If we want to **deviate from the argument order** of the *Usage* category of a function's help document in our function call, we have to **state the argument names** as well.

```
seq(to = 20, by = -5, length.out = 5)
```

```
## [1] 40 35 30 25 20
```

By **not setting an argument** explicitly, we use the function's **default value for this argument** (here: a starting value of 1).

```
seq(to = 5, by = 1)
```

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

Repetition of patterns in vectors

Sometimes we have to repeat certain patterns in a vector, e.g. when adding a month indicator to daily data.

```
rep(29, times = 4)
## [1] 29 29 29 29
rep(c(2, 9), times = 4)
## [1] 2 9 2 9 2 9 2 9
rep(c(2, 9), each = 4)
## [1] 2 2 2 2 9 9 9 9
rep(c(2, 9), times = 3, each = 3)
##  [1] 2 2 2 9 9 9 2 2 2 9 9 9 2 2 2 9 9 9
```


Vectorisation

R is a vectorised language. This means that many functions work on vectors as well as on scalars and that you can avoid `for` loops in many instances. As R is an interpreted language, loops are rather slow in R. So you save time by using vectorisation in your code (more on that later).

```
0:10 + 10:0
##  [1] 10 10 10 10 10 10 10 10 10 10 10
sqrt(c(1, 16, 36, 144))
## [1]  1  4  6 12
```

Vectorised functions are basically wrappers for functions written in C, C++, FORTRAN, etc. Computations are done element-wise.

ATTENTION:
Not all functions are vectorised.

Recycling

When performing operations on vectors of differing lengths, R will recycle the shortest vector as needed, i.e. until the length of the longest vector is matched.

```
1:10 + 10
##  [1] 11 12 13 14 15 16 17 18 19 20
1:10 + 1:5
##  [1]  2  4  6  8 10  7  9 11 13 15
1:4 + 1:3
## Warning in 1:4 + 1:3: longer object length is not a multip
## length
## [1] 2 4 6 5
```

ATTENTION:
Always be aware of recycling, as it might conceal problems.

Assignments I

You can assign an object (like a vector) to a name and use this name afterwards to refer to the object.

In R you should use the assignment operator <- to do the assignment. Although you could also use =, the other operator is the preferred option as it visually distinguishes assignments from the setting of argument values and guarantees compatibility with some older functions written in S.

Assignments II

```
x <- c(20, 19, 25, 1)
x
## [1] 20 19 25  1
y <- c(9, 10, 4, 28)
(z <- x + y)
## [1] 29 29 29 29
```

Note that an assignment is *silent*, i.e. the object will not be printed, unless it is enclosed by parentheses (like in the fourth input code line).

Assignments III

There are very few limitations as far as the choice of object names is concerned. However, there are some rules you **have to follow**:

1. You can use letters, numbers, underscores, and periods.
2. Names have to start with a letter or a period.
3. If the name starts with a period, it cannot continue with a number.
4. You cannot use any of the following names:
   break, else, FALSE, for, function, if, Inf, NA, NaN, next, repeat, return, TRUE, while.

There are also some suggestions you **should follow**:

1. Names should not coincide with existing function names.
2. Names should describe data, functions, etc.

ATTENTION:
R is case-sensitive.

Global environment

We already generated three named objects (vectors). These were generated within the global environment, which might be best thought of as our R *desktop*.

To see the current contents of the global environment, we can either look into the *Environment* tab in pane 4 or we can use the function `ls`.

```
ls()
```

```
## [1] "x" "y" "z"
```

The tab in pane 4 does not only provide the names of the objects but some additional information that we will understand later.

Saving objects

We can easily save one or multiple objects in an `RData` file that is strongly compressed to a directory (working directory per default).

```
x
## [1] 20 19 25  1
save(x, file = "x.RData")
save(x, y, z, file = "Vectors.RData")
```

Removing and loading objects

If we do not need certain objects in the current session anymore and would like to declutter our global environment and/or free up some RAM, we can remove objects. If we need these objects again later, we can just load them.

```
rm(x)
ls()
## [1] "y" "z"
load("x.RData")
ls()
## [1] "x" "y" "z"
```


Loading *unknown* objects

Sometimes you have to work with objects generated by other people. They might not name the `RData` file like the object it contains. Finding the correct object in your R session may then become tiresome if your global environment is already crowded. However, you can directly assign the object to a new name.

```
some_name <- rep(1, 4)
save(some_name, file = "Some_other_name.RData")
new_name <- get(load("Some_other_name.RData"))
new_name
## [1] 1 1 1 1
ls()
## [1] "new_name"  "some_name" "x"         "y"         "z"
```

Note that you now have a duplicate of the respective object in your global environment.

Starting from scratch

If you need to clean your whole *desktop* up and start from scratch, you can remove all objects generated and/or loaded during the current R session.

```
# Do not run: rm(list = ls())
```


Sorting vectors

We can sort vectors, either in ascending or descending order.

```
sort(x)
## [1]  1 19 20 25
sort(x, decreasing = TRUE)
## [1] 25 20 19  1
```

The index

If we print a vector to the console, the number in square brackets indicates the index of the vector element to the right of it.

```
some_vector <- c(seq(2, 20, 2), 1:10, seq(3, 30, 3))
some_vector
##  [1]  2  4  6  8 10 12 14 16 18 20  1  2  3  4  5  6  7  8
## [26] 18 21 24 27 30
```

In this example the first element of vector `some_vector` is 2 and the 26th element is 18.

ATTENTION:
Although vectors look like row vectors in the console, they actually are column vectors.

order

`order` gives back the positions of the elements of the sorted vectors in the original (unsorted) vector.

```
order(x)
## [1] 4 2 1 3
order(x, decreasing = TRUE)
## [1] 3 1 2 4
```


Indexing vectors I

We can index vectors to extract elements from them using square brackets. *Positive* indices extract the respective elements, while *negative* indices exclude the respective elements and extract the remaining elements.

```
some_vector[] # complete vector
##  [1]  2  4  6  8 10 12 14 16 18 20  1  2  3  4  5  6  7  8
## [26] 18 21 24 27 30
some_vector[1]
## [1] 2
some_vector[1:10]
##  [1]  2  4  6  8 10 12 14 16 18 20
some_vector[-(1:20)]
##  [1]  3  6  9 12 15 18 21 24 27 30
some_vector[c(5, 10, 15)]
## [1] 10 20  5
```


Indexing vectors II

We can repeatedly extract elements of vectors and we can index within extracted elements of a vector.

```
some_vector[c(rep(5, 5), rep(1, 5))]
```

```
##  [1] 10 10 10 10 10  2  2  2  2  2
```

```
some_vector[c(1, 5, 10)][3] # third element of new vector
```

```
## [1] 20
```

Lengths, sums, products, and means of vectors

We can easily extract information from (numeric) vectors.

```
a <- c(10, 2, 20)
length(a)
## [1] 3
sum(a)
## [1] 32
cumsum(a)
## [1] 10 12 32
prod(a)
## [1] 400
cumprod(a)
## [1]  10  20 400
mean(a)
## [1] 10.66667
```


Minima, maxima, and ranges of vectors

```
min(a)
## [1] 2
cummin(a)
## [1] 10  2  2
max(a)
## [1] 20
cummax(a)
## [1] 10 10 20
range(a)
## [1]  2 20
```


Rounding of vectors I

We can round numeric vectors as needed.

```
b <- c(pi, 2.71828, 6.52525)
round(b)
## [1] 3 3 7
round(b, 2)
## [1] 3.14 2.72 6.53
round(b, -1)
## [1]  0  0 10
floor(b)
## [1] 3 2 6
ceiling(b)
## [1] 4 3 7
```


Rounding of vectors II

```
signif(1251.48, 5)
## [1] 1251.5
signif(1251.48, 4)
## [1] 1251
signif(1251.48, 3)
## [1] 1250
trunc(c(-2020.8, 2020.8))
## [1] -2020  2020
```

Overview

The following table lists the data types that you will encounter in R. You can query the data type of an object by providing its name as an argument to the function `mode`.

Data type	Example	Output of `mode`
Empty set ($\emptyset$)	`NULL`	`"NULL"`
Logical values	`TRUE`	`"logical"`
Integers and real numbers	`1.25`	`"numeric"`
Complex numbers	`2+1i`	`"complex"`
(Strings of) characters	`"Hello world!"`	`"character"`

The information content of the shown data types increases with every line.

Information content I

```
l_part <- c(TRUE, FALSE)
mode(l_part)
## [1] "logical"
r_part <- 1.25
mode(r_part)
## [1] "numeric"
(c1 <- c(l_part, r_part)) # TRUE: 1, FALSE: 0
## [1] 1.00 0.00 1.25
mode(c1)
## [1] "numeric"
```

Note that the new *combined* vector `c1` has been coerced to type numeric. R does this because numeric vectors convey more information than logical vectors. R *keeps* the data type with the largest information content. Therefore, you cannot *mix* data types within a vector.

Information content II

```
(c_part <- complex(real = -1, imaginary = 3))
## [1] -1+3i
mode(c_part)
## [1] "complex"
(c2 <- c(c1, c_part))
## [1]  1.00+0i  0.00+0i  1.25+0i -1.00+3i
mode(c2)
## [1] "complex"
ch_part <- "Abc"
(c3 <- c(c2, ch_part)) # Note the double quotes
## [1] "1+0i"    "0+0i"    "1.25+0i" "-1+3i"   "Abc"
mode(c3)
## [1] "character"
```

Character vectors can represent vectors of all *lower* data types as well.

Information content III

R's information preserving behaviour has its drawback.

```
c3[1] + c3[3]
```

```
## Error in c3[1] + c3[3]: non-numeric argument to binary oper
```

Information content IV

`numeric(2)` creates an *empty* numeric vector containing only zeros. The functions named after the different remaining data types work the same way.

Functions starting with an `as.` coerce vectors into specified data types.

We can transfer *higher* data types into *lower* data types, albeit with a loss of information.

```
two_numbers <- numeric(2)
two_numbers[] <- c(0, 5) # overwriting previous elements
as.logical(two_numbers) # 5 is not 0 and must then be 1
## [1] FALSE  TRUE
as.numeric(complex(real = 2, imaginary = 1))
## Warning: imaginary parts discarded in coercion
## [1] 2
```


Logical operators I

Using logical operators, we can transform other vectors to logical vectors by checking whether certain conditions are met. We will see at a later stage that this is useful for indexing objects, doing basic computations, and control flow.

```
b
## [1] 3.141593 2.718280 6.525250
b == pi
## [1]  TRUE FALSE FALSE
b != pi # ! always used for negation
## [1] FALSE  TRUE  TRUE
b < 3
## [1] FALSE  TRUE FALSE
b[1] >= 3
## [1] TRUE
```


Logical operators II

We can combine conditions using logical and (`&`), logical and/or (`|`), and logical or, the latter using the function `xor`.

```
b[1] > 3 & b[3] > 10 # TRUE if both are TRUE
## [1] FALSE
b[1] < 3 | b[3] < 10 # TRUE if at least one is TRUE
## [1] TRUE
xor(b[1] > 3, b[3] < 10) # TRUE if only one is TRUE
## [1] FALSE
```


Logical operators III

There are also non-vectorised versions (`&&` and `||`, respectively) that only evaluate the first elements of vectors (left to right). These should only be used if one is sure that the vector lengths are actually 1. Otherwise, we can use `all` and `any`, both returning a vector of length 1.

```
all(b > 1)
## [1] TRUE
any(b > 20)
## [1] FALSE
```

Computations using logical vectors

How many multiples of 3 do the natural numbers from 1 through 1,000 contain?

```
sum(1:1000 %% 3 == 0)
```

```
## [1] 333
```


Locating certain elements within a vector

We can use the function `which` in combination with logical operators to find the positions of certain elements within a vector.

```
(check <- b != pi)
## [1] FALSE  TRUE  TRUE
b[check] # only elements that are TRUE extracted
## [1] 2.71828 6.52525
(1:length(b))[check]
## [1] 2 3
which(b != pi)
## [1] 2 3
which.min(b)
## [1] 2
which.max(b)
## [1] 3
```


Naming vectors

You can name vectors using character vectors and use the names to index the original vector.

```
names(b) <- c("Pi", "e", "Six_and_change")
b[c("Pi", "Six_and_change")]
##             Pi Six_and_change
##       3.141593       6.525250
```


T vs. TRUE

Try to avoid using `T` instead of `TRUE` and `F` instead of `FALSE`.

```
TRUE & TRUE
## [1] TRUE
T & T
## [1] TRUE
T <- 0
T & T
## [1] FALSE
```

Testing for exact equality

== only compares values, while `identical` also compares data types.

Functions starting with an `is.` test for specified data types.

```
number1 <- -1
number2 <- -1^3
number1 == number2
## [1] TRUE
identical(number1, number2)
## [1] TRUE
is.numeric(c(number1, number2))
## [1] TRUE
1 == as.integer(1)
## [1] TRUE
identical(1, as.integer(1))
## [1] FALSE
```


Matrices I

Adding a second dimension, we move from vectors to matrices.
Using the function `matrix`, we can define the data vector using the argument `data` as well as the number of rows and columns, using the arguments `nrow` and `ncol`, respectively.

```
(m1 <- matrix(data = 1:16, nrow = 4))
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

ATTENTION:
By default, matrices are *filled* by column.

Matrices II

Just like with vectors, basic arithmetic operations are performed vector-wise (i.e. first elements, second elements, ...).

```
m2 <- matrix(16:1, ncol = 4)
m1 + m2
##      [,1] [,2] [,3] [,4]
## [1,]   17   17   17   17
## [2,]   17   17   17   17
## [3,]   17   17   17   17
## [4,]   17   17   17   17
```


Matrices III

Matrices may be filled by row as well.
Or we just transpose the *original* matrix.

```
(m3 <- matrix(1:16, nrow = 4, byrow = TRUE))
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
t(m1)
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
```

Indexing matrices I

We can still use *vector indexing* with matrices.
But we can now also use row and/or column indices, located before and after a comma in the square brackets, respectively (function arguments).

```
m3[9] # default column-wise orientation
## [1] 3
m3[1, 3]
## [1] 3
m3[2, -1] # from matrix to vector
## [1] 6 7 8
m3[2, -1, drop = FALSE]
##      [,1] [,2] [,3]
## [1,]    6    7    8
```

By default R drops *superfluous* dimensions.

Indexing matrices II

Leaving the row index empty leads to an extraction of every row of the desired columns.
Leaving the column index empty leads to an extraction of every column of the desired rows.

```
m3[, 1:2]
##      [,1] [,2]
## [1,]    1    2
## [2,]    5    6
## [3,]    9   10
## [4,]   13   14
m3[3:4, ]
##      [,1] [,2] [,3] [,4]
## [1,]    9   10   11   12
## [2,]   13   14   15   16
```


Manipulating matrix elements

Like with vectors, we can combine indexing and assignments to overwrite elements of a matrix.

```
to_manipulate <- seq(4, by = 4, length.out = 4)
m3[to_manipulate] <- 100 # recycling
m3
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]  100  100  100  100
```

ATTENTION:
There is no warning issued when elements are overwritten.

Appending rows and columns to matrices

We can easily append rows and columns to matrices.

```
(m4 <- matrix(1:4, ncol = 2))
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
rbind(m4, c(3, 5))
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
## [3,]    3    5
cbind(m4, c(5, 6))
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Matrix multiplication

We need a special operator to do matrix multiplication.

```
(m5 <- matrix(1:6, ncol = 3))
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
(m6 <- matrix(10:5, nrow = 3))
##      [,1] [,2]
## [1,]   10    7
## [2,]    9    6
## [3,]    8    5
m5 %*% m6
##      [,1] [,2]
## [1,]   77   50
## [2,]  104   68
```


Finding the inverse of a matrix

Finding the inverse of a matrix (if it exists) is easy in R. We can use the function `solve` that usually solves linear equation systems, but only passing the first argument.

```
(m7 <- matrix(c(1, 2, 2, 3), ncol = 2))
##      [,1] [,2]
## [1,]    1    2
## [2,]    2    3
solve(m7)
##      [,1] [,2]
## [1,]   -3    2
## [2,]    2   -1
m7 %*% solve(m7)
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```


Eigenvalues and eigenvectors

We can find the eigenvalues and eigenvectors of matrices using `eigen`. The output is a data structure that we will learn about in the following chapter (a list), including how we can index it.

```
eigen(m7)
## eigen() decomposition
## $values
## [1]  4.236068 -0.236068
##
## $vectors
##           [,1]       [,2]
## [1,] 0.5257311 -0.8506508
## [2,] 0.8506508  0.5257311
```


Useful matrix functions I

There are some useful functions to analyse matrices.

```
dim(m5)
## [1] 2 3
nrow(m5)
## [1] 2
ncol(m5)
## [1] 3
length(m5) # matrix as vector
## [1] 6
rowSums(m5)
## [1]  9 12
colSums(m5)
## [1]  3  7 11
```

Useful matrix functions II

```
diag(m3)
## [1]   1   6  11 100
diag(m3) <- 3
m3
##      [,1] [,2] [,3] [,4]
## [1,]    3    2    3    4
## [2,]    5    3    7    8
## [3,]    9   10    3   12
## [4,]  100  100  100    3
diag(2)
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```


Still, one problem remains...

As with vectors, we cannot mix data types in matrices.

```
colnames(m5) <- c("Variable1", "Variable2", "Variable3")
ID <- c("I01", "I02")
(m5_new <- cbind(m5, ID))
##      Variable1 Variable2 Variable3 ID
## [1,] "1"       "3"       "5"       "I01"
## [2,] "2"       "4"       "6"       "I02"
mean(m5_new[, "Variable1"])
## Warning in mean.default(m5_new[, "Variable1"]): argument is
## logical: returning NA
## [1] NA
```


Arrays I

If we do not need different data types in our object, but more than two dimensions, we can work with arrays. A two-dimensional array is simply a matrix. A one-dimensional array is essentially a vector. However, it might be treated differently by some functions.

As an example of a three-dimensional array, we use an example data set included in R's base installation. We can just send the data set name `HairEyeColor` to the console to print it and we can use `?HairEyeColor` to get some basic information about the data set.

Arrays II

```
HairEyeColor
## , , Sex = Male
##
##        Eye
## Hair    Brown Blue Hazel Green
##   Black    32   11    10     3
##   Brown    53   50    25    15
##   Red      10   10     7     7
##   Blond     3   30     5     8
##
## , , Sex = Female
##
##        Eye
## Hair    Brown Blue Hazel Green
##   Black    36    9     5     2
##   Brown    66   34    29    14
```

Indexing arrays I

`HairEyeColor` contains matrices of the absolute frequencies of cross-combinations of hair colours (rows) and eye colours (columns) in male and female (*matrices*) students.

As before, within square brackets row indices are passed as the first argument and column indices as the second argument. The next dimension (here: matrix indices) is passed as the third argument and so on.

```
dim(HairEyeColor)
## [1] 4 4 2
HairEyeColor["Brown", "Hazel", "Female"]
## [1] 29
HairEyeColor["Blond", , "Female"]
## Brown  Blue Hazel Green
##     4    64     5     8
```


Indexing arrays II

R *drops dimensions* here by default as well. If needed, we can always set argument `drop` to `FALSE`.

```
HairEyeColor["Red", , ]
##        Sex
## Eye     Male Female
##   Brown   10     16
##   Blue    10      7
##   Hazel    7      7
##   Green    7      7
mode(HairEyeColor)
## [1] "numeric"
HairEyeColor[1, 1, 1] <- "50"
mode(HairEyeColor)
## [1] "character"
```


Generation of arrays I

We just saw that the *old* problems remain: we need conforming structures in the array (like the 4x4 matrices here) and we cannot mix data types.

To generate arrays, we can either use the function `array` or first generate the data vector and then manipulate its dimensions.

```
an_array <- array(1:20, dim = c(2, 5, 2))
data_vector <- 1:20
dim(data_vector) <- c(2, 5, 2)
all(an_array == data_vector)
## [1] TRUE
```


Generation of arrays II

```
an_array
## , , 1
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
##
## , , 2
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   11   13   15   17   19
## [2,]   12   14   16   18   20
```

Statistical Programming with R

Chapter 3 - Lists and data frames

Florian Ertz, Jan Pablo Burgard, and Ralf Münnich

Economic and Social Statistics
Economics / Faculty IV
Trier University

Winter semester 2021/2022

Copyright statement

A flexible data structure

Earlier on, we saw that vectors, matrices, and arrays are not very flexible data structures. We were not able to mix data types in these structures. Additionally, we were forced to *fill* matrices.

The list is a different data structure that is very flexible.

We will first generate a few list elements of different data types and lengths/dimensions. A list can accommodate these.

```
a_num_vector <- 1:10
a_chr_vector <- "A single element"
a_matrix <- matrix(1:6, nrow = 2, ncol = 3)
```

Generation of a list

We now construct a list using the function `list`, also naming its elements.

```
(list1 <- list(Numeric_vector = a_num_vector,
               Character_vector = a_chr_vector,
               Matrix = a_matrix
               ))
## $Numeric_vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $Character_vector
## [1] "A single element"
##
## $Matrix
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```


Indexing a list I

We can address single list elements in three different ways.

```
list1[1]
## $Numeric_vector
##  [1]  1  2  3  4  5  6  7  8  9 10
is.list(list1[1])
## [1] TRUE
list1[[1]] # Note the double square bracket
##  [1]  1  2  3  4  5  6  7  8  9 10
is.list(list1[[1]])
## [1] FALSE
list1[["Numeric_vector"]]
##  [1]  1  2  3  4  5  6  7  8  9 10
list1$Numeric_vector
##  [1]  1  2  3  4  5  6  7  8  9 10
```


Indexing a list II

Indexing a list with a **single square bracket** gives back a **list**.
This should be done when multiple elements are to be addressed or when a list is required as input later in the code.

Indexing a **single list element** with a **double square bracket** or the **dollar sign and** the **name** of the list element gives back an object in the list element's original data structure (like the vector on the previous slide). This is the **preferred option in most cases**.

Lower level indexing (i.e. within list elements) works like before. We just append it to the *list indexing*.

To get the names of list elements and change them (in combination with an assignment), we can use `names`.

```
names(list1)
## [1] "Numeric_vector"   "Character_vector" "Matrix"
```


Indexing a list III

```
list1[1][[1]][1:5] # first element of list with one element
## [1] 1 2 3 4 5
list1[[1]][1:5]
## [1] 1 2 3 4 5
list1$Numeric_vector[1:5]
## [1] 1 2 3 4 5
list1$Matrix[2, 2:3, drop = FALSE]
##      [,1] [,2]
## [1,]    4    6
list1[[c(1, 10)]] # recursive indexing, not recommended
## [1] 10
```

Recursive lists I

We can also use recursive lists, i.e. lists within lists within lists...

```
parent_list <- list(Some_list = list1,
                    Another_list = list(A = 10:1,
                                        B = 1:10
                                        )
                    )
grandparent_list <- list(First = parent_list,
                         Second = "Useless_comment"
                         )
```


Recursive lists II

We can look into the internal structure of any R object by using the function `str`.

```
str(grandparent_list)
## List of 2
##  $ First :List of 2
##   ..$ Some_list   :List of 3
##   .. ..$ Numeric_vector  : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ Character_vector: chr "A single element"
##   .. ..$ Matrix          : int [1:2, 1:3] 1 2 3 4 5 6
##   ..$ Another_list:List of 2
##   .. ..$ A: int [1:10] 10 9 8 7 6 5 4 3 2 1
##   .. ..$ B: int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ Second: chr "Useless_comment"
```


Recursive lists III

Getting to the entry in the second row and third column of the matrix in our original list (`list1`) requires some longer indexing.

```
grandparent_list$First$Some_list$Matrix[2, 3]
## [1] 6
grandparent_list[[1]][[1]][[3]][2, 3]
## [1] 6
```

ATTENTION:
Although using names in indexing leads to *longer* code, it is less error-prone, especially if the list is regularly modified.

Appending and *shrinking* lists

Using indexing and assignments, it is easy to either append new elements to a list or to *delete* elements of a list.

```
list_copy <- grandparent_list
list_copy$Third <- array(8:1, dim = c(2, 2, 2))
list_copy$First <- NULL
str(list_copy)
## List of 2
##  $ Second: chr "Useless_comment"
##  $ Third : int [1:2, 1:2, 1:2] 8 7 6 5 4 3 2 1
```

Operations on several list elements

We might want to use certain operations on every element of a list or on some of them. Doing this separately for every element would be inefficient (more on efficiency later). But there is a way to avoid this.

Below, we want to sum all elements of the vectors in `Another_list`, which is on the second level of `grandparent_list`.

```
lapply(grandparent_list$First$Another_list, sum)
## $A
## [1] 55
##
## $B
## [1] 55
sapply(grandparent_list$First$Another_list, sum)
##  A  B
## 55 55
```


lapply and sapply I

We just saw that the outputs of the functions `lapply` and `sapply`, both taking a list as their first argument and a function name as their second argument, differ. While `lapply` keeps the original list structure in its output, `sapply` tries to *simplify* the output; a behaviour much like the *dropping of dimensions* we saw earlier.

```
sapply(grandparent_list$First$Another_list, range)
##      A  B
## [1,] 1  1
## [2,] 10 10
```


lapply and sapply II

If the function outputs do not allow for a simplification, however, the outputs of `lapply` and `sapply` are the same.

```
list2 <- list(V1 = rep(1, 10),
              V2 = rep(1:2, 5),
              V3 = rep(1:5, 2)
              )
sapply(list2, unique) # unique values in objects
## $V1
## [1] 1
##
## $V2
## [1] 1 2
##
## $V3
## [1] 1 2 3 4 5
```


Anonymous functions

At one point or another, you will surely have to use operations that are not covered by existing functions, as they are tailored to a very specific end. You could then program your own function (discussed at a later stage) and refer to its name. Sometimes, though, this might not be worthwhile because the function would not be special enough. In such cases, you want to use anonymous functions.

```
sapply(list2, function(x) x[1] + x[2] - x[5])
## V1 V2 V3
##  1  2 -2
```

Here, `function(x)` tells R that a function with one argument will be defined in what follows. Here, `x` sequentially represents every list element, i.e. vector `V1` first, then vector `V2`, and finally vector `V3`. This is why we can use vector indexing.

mapply

If we need to combine information from elements of separate lists, we can use `mapply`, whose argument order differs from the one found in `lapply`.

```
list_a <- list(a1 = c(120, 80, 70, 160, 130),
               a2 = c(1000, 300, 400, 800, 200)
               )
list_b <- list(b1 = 2, b2 = 5)
mapply(function(x, y)
       x[y], # 2nd element of a1, 5th element of a2
       list_a, # x is list_a
       list_b, # y is list_b
       SIMPLIFY = TRUE # avoiding output of list
       )
##  a1  a2
##  80 200
```


Data frames

A data frame is a special case of a list, where all list elements are vectors of the same length and the list is printed and can be indexed like a matrix. As we are still dealing with a list, we can now mix data types.

This feature makes the data frame the predominant data structure for data sets in R. To give but one example, in the economic and social sciences we are often dealing with data on individuals that contain metric variables (e.g. income) as well as non-metric variables (e.g. marital status).

In every data frame, which is basically a rectangular organisation of data, cases/observations are in rows and variables are in columns.

Many functions expect data frames as input.

Generation of data frames

To generate a data frame from scratch, we use `data.frame`.

```
(df1 <- data.frame(
  Income = c(1300, 3500, 900, 6800, 2700),
  Age = c(21, 30, 16, 70, 35),
  Comment = c("None", "None", "None", "Error?", "None"),
  Imputed = c(rep(FALSE, 4), TRUE)
  ))
##   Income Age Comment Imputed
## 1   1300  21    None   FALSE
## 2   3500  30    None   FALSE
## 3    900  16    None   FALSE
## 4   6800  70  Error?   FALSE
## 5   2700  35    None    TRUE
```


Indexing of and computations with data frames

We can now index the data frame like a matrix and we can do computations on its numeric elements, although one *column* contains character strings and another logical values.

```
str(df1)
## 'data.frame':    5 obs. of  4 variables:
##  $ Income : num  1300 3500 900 6800 2700
##  $ Age    : num  21 30 16 70 35
##  $ Comment: chr  "None" "None" "None" "Error?" ...
##  $ Imputed: logi  FALSE FALSE FALSE FALSE TRUE
df1[3, "Age"]
## [1] 16
mean(df1$Income)
## [1] 3040
```

Subsetting data frames

We can easily subset data frames using `subset`.

```
subset(df1, Imputed == TRUE)
##   Income Age Comment Imputed
## 5   2700  35    None    TRUE
```


Factors I

Factors can be used to group data and are very useful when it comes to the analysis of subgroups within data sets. They map indices/levels to labels. As we will see later, many functions use factors.

An easy way to go from numeric vectors to factor vectors is to use `cut`.

```
df1$Age_group <- cut(
  df1$Age,
  breaks = c(0, 25, 50, 100),
  labels = c("Young", "Middle", "Old"),
  right = TRUE # intervals closed on the right
  )
df1$Age_group
## [1] Young  Middle Young  Old    Middle
## Levels: Young Middle Old
```


Factors II

```
str(df1)
## 'data.frame':    5 obs. of  5 variables:
##  $ Income   : num  1300 3500 900 6800 2700
##  $ Age      : num  21 30 16 70 35
##  $ Comment  : chr  "None" "None" "None" "Error?" ...
##  $ Imputed  : logi  FALSE FALSE FALSE FALSE TRUE
##  $ Age_group: Factor w/ 3 levels "Young","Middle",..: 1 2 1
```


Factors III

Below, we construct a factor vector using `factor`. Here, we could provide levels and labels explicitly using the respective arguments if we wish.

```
as.numeric(df1$Age_group)
## [1] 1 2 1 3 2
numbers <- c(3, 10, 1, 3, 7, 7, 5, 1)
a_factor <- factor(numbers)
as.numeric(a_factor) # 3 is 2nd level, 10 is 5th level etc.
## [1] 2 5 1 2 4 4 3 1
as.numeric(as.character(a_factor))
## [1]  3 10  1  3  7  7  5  1
```

ATTENTION:
Be careful when you transfer factors with numbers as labels back to numeric vectors.

Factors IV

```
some_codes <- c(3, 2, 3, 3, 1, 1, 2)
factor(some_codes,
       levels = 1:3,
       labels = c("Low", "Medium", "High")
       )
## [1] High   Medium High   High   Low    Low    Medium
## Levels: Low Medium High
```


First glances at data frames I

To look at some useful functions, we now work with another example data set from R's base installation, `ToothGrowth`. You find the details of the data set using `?ToothGrowth`.

To show only the first or last cases/elements (six by default) in a data frame/vector, we use `head` and `tail`, respectively.

```
head(ToothGrowth) # First six cases
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```


First glances at data frames II

```
tail(ToothGrowth, 3) # Last three cases
##     len supp dose
## 58 27.3   OJ    2
## 59 29.4   OJ    2
## 60 23.0   OJ    2
length(ToothGrowth)
## [1] 3
```

Note that `length` also gives you the number of elements of a list. With a data frame, each column is an element.

Going to a *proper* list

If we need to go from a data frame to a *proper* list, we can `split` a data frame into a list with as many elements as the number of levels of the factor (combination) that we use to split the data frame.

```
proper_list <- split(ToothGrowth, ToothGrowth$supp)
str(proper_list) # Notice the list element names
## List of 2
##  $ OJ:'data.frame':  30 obs. of  3 variables:
##   ..$ len : num [1:30] 15.2 21.5 17.6 9.7 14.5 10 8.2 9.4 1
##   ..$ supp: Factor w/ 2 levels "OJ","VC": 1 1 1 1 1 1 1 1 1
##   ..$ dose: num [1:30] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
##  $ VC:'data.frame':  30 obs. of  3 variables:
##   ..$ len : num [1:30] 4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.
##   ..$ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2
##   ..$ dose: num [1:30] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
```

Addressing data frame elements I

R will not have the element names of a data frame in its name space by default, so we cannot address them *directly*. However, we can add them to the name space.

```
len
## Error in eval(expr, envir, enclos): object 'len' not found
attach(ToothGrowth)
head(len)
## [1]  4.2 11.5  7.3  5.8  6.4 10.0
len[1] <- 50
head(ToothGrowth, 1); detach(ToothGrowth) # separate calls
##   len supp dose
## 1 4.2   VC  0.5
```

ATTENTION:
Changes to *new* vectors will not be applied to the data frame.

Addressing data frame elements II

Another option to work with certain elements/variables of a data frame is to use `with`.

```
with(ToothGrowth, c(max(len), max(dose)))
## [1] 33.9  2.0
```


`order` with multiple arguments

We can now use `order` with more than one argument. Ties in the first argument will be broken by the second argument, etc.

```
tg_ordered <-
  ToothGrowth[order(ToothGrowth$len, ToothGrowth$dose), ]
tg_ordered[14:19, ]
##     len supp dose
## 8  11.2   VC  0.5
## 2  11.5   VC  0.5
## 17 13.6   VC  1.0
## 35 14.5   OJ  0.5
## 18 14.5   VC  1.0
## 49 14.5   OJ  1.0
```


Extracting specific cases based on one variable

We can, e.g., only extract cases (rows) where the variable `len` takes on a value of either 5.8, 14.5, or 27.3.

```
ToothGrowth[ToothGrowth$len %in% c(5.8, 14.5, 27.3), ]
##     len supp dose
## 4   5.8   VC  0.5
## 18 14.5   VC  1.0
## 35 14.5   OJ  0.5
## 49 14.5   OJ  1.0
## 50 27.3   OJ  1.0
## 58 27.3   OJ  2.0
```

Note that `A %in% B` creates a logical vector of a length equal to the length of `A`, where every entry whose corresponding element of `A` is also an element of `B` is set to `TRUE`.

Statistical Programming with R

Chapter 4 - Packages

Florian Ertz, Jan Pablo Burgard, and Ralf Münnich

Economic and Social Statistics
Economics / Faculty IV
Trier University

Winter semester 2021/2022

Copyright statement

Packages

As discussed in the first chapter, R is not a program in the traditional sense. Each user can customise her R installation by installing extensions in addition to the base installation. These extensions are called packages and usually contain the following things:

1. Functions
2. Documentation
3. Data sets

Typically, individual packages deal with very specific topics, ranging from data visualisation through data manipulation to certain statistical methods.

Default packages

As of October 2020, the following packages are loaded per default in every R session:

1. `base`
2. `datasets`
3. `graphics`
4. `grDevices`
5. `methods`
6. `stats`
7. `utils`

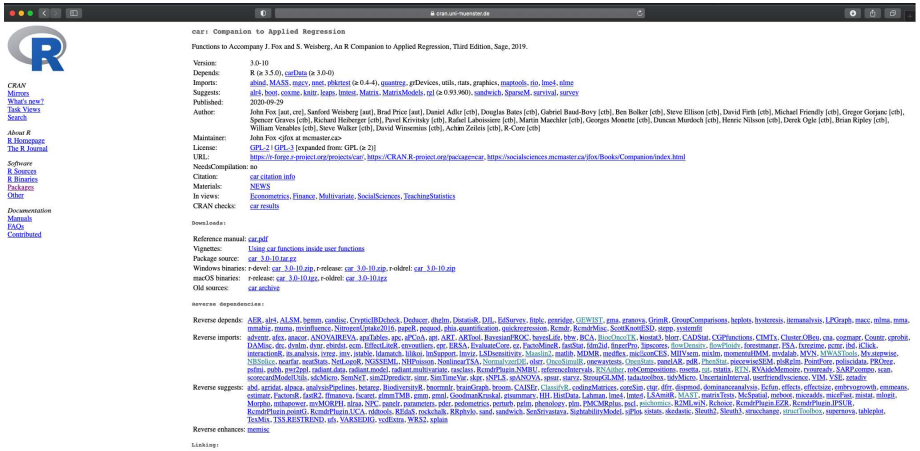
You can check which packages are loaded in pane 3 in the tab *Packages*.

CRAN

There is a list of quality-controlled packages contributed to the community on the Comprehensive R Archive Network (CRAN), the major R repository. On your chosen mirror, you will find it when you follow the link *Packages* on the left-hand navigation bar. You can either browse the packages by date of publication or by name.

By clicking a package name in one of these lists, you will reach the website of the respective package. In addition to other detailed information, source files, and suggested citation, you find the reference manual, which is a document that contains documentation for all functions and data sets the package contains, and sometimes also a vignette, a small paper on the usage of core package functionality.

`car` package website



Package installation from CRAN

Before starting the installation of packages, it might be worthwhile to set a preferred mirror in RStudio to prevent the respective dialogue from popping up before every installation. You can do so here:

Tools → **Global Options...** → **Packages** → **Change...**

We will now install the package `fortunes` using the function `install.packages` and providing the package name as a character string. We could also provide a character vector to install several packages at once.

ATTENTION:
You have to execute this call. We have only commented it out here because we are using *R Markdown*.

```
#install.packages("fortunes")
```

Loading a package

To be able to use functions from a specific package in an R session, we have to load this package first using `library`. Note that the package name is now provided without quotation marks; we are dealing with an R object now. While we could `detach` a package later on, this is usually not needed.

After this is done, we can use package functions, like `fortune` (no plural) here.

Please check `?fortune` first.

```
library(fortunes)
fortune(239) # use fortune() for random quote
##
## Yes we CRAN!
##     -- Aurelien Latouche (suggestion for a T-shirt slogan)
##        private communication (July 2009)
```


Addressing specific functions

Sometimes different packages contain a function of the same name. In such a case, it is sensible to use the following way to call the function, where we combine the package name and the function name.

To the best of our knowledge, there is no other function named `fortune`, though.

```
fortunes::fortune(214)
##
## R may be the wrong tool for the job, but it's the wrong jol
##     -- Rolf Turner (about solving 100th degree polynomials)
##        R-help (May 2008)
```


Updating packages

As most packages on CRAN are regularly updated, you should update your local package installations as well.

Using `update.packages()`, you will be asked for every package that has a newer version on CRAN whether it should be updated individually. The option below just updates without asking first.

ATTENTION:
Do not run the code below now or when you have work to do in R. It might take a while...

```
# Again, do not run now: update.packages(ask = FALSE)
```

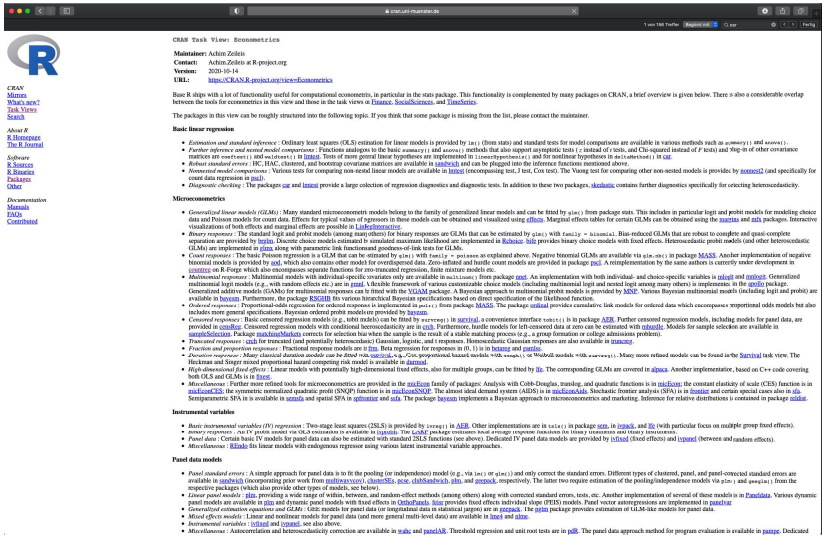

Task Views

The CRAN Task Views offer curated lists of the most-used packages in different disciplines like, e.g., clinical trials, econometrics, or time series analysis.

Within a short descriptive text, you find links to the listed packages there.

To browse the Task Views and get instructions on how to install complete Task Views, just follow the link *CRAN Task Views* on the *Packages* website on your preferred mirror.

Task View - Econometrics



For the xkcd afficionados...

```
#install.packages("RXKCD")
library(RXKCD)
getXKCD(302)
```

Source: xkcd.com (2020).

Statistical Programming with R

Chapter 5 - Visualisation

Florian Ertz, Jan Pablo Burgard, and Ralf Münnich

Economic and Social Statistics
Economics / Faculty IV
Trier University

Winter semester 2021/2022

Copyright statement

Three systems

Visualisation is one of the major fortes of R. You can try `demo(graphics)` to get a first idea.

There are three systems for producing graphs in R:

1. `graphics`
2. `lattice`
3. `ggplot2`

In this first subsection, we concentrate on base graphs that can be produced with tools found in the pre-installed and pre-loaded package `graphics`.

The `plot` function I

Usually, the `plot` function is the hub for starting the production of base graphs in R, originating from a scatterplot.

It is a so-called generic function, meaning that it employs a specific variant of the more general function that is specifically suited for the data input provided by the user *under the hood*.

When we start a new plot, we are doing high-level plotting. Think of this as the initial sketch of a painting on a blank canvas.

The `plot` function II

To position points in the plot region which is bounded by the rectangle defined by the axes, a Cartesian coordinate system is used.

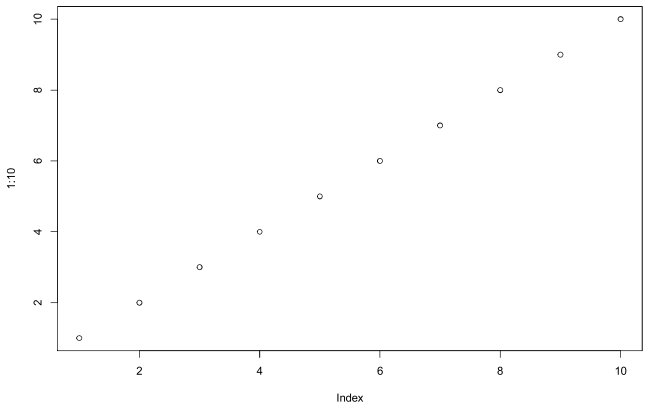
If you **only provide** a vector to **argument** `x`, this vector will be taken as the y-coordinates, while a simple index (`1:length(x)`) will be taken as the x-coordinates.

If you **provide** vectors to **both argument** `x` **and argument** `y`, these will be taken as the x- and y-coordinates, respectively.

The default colour for plotting is black. More on colours in R at a later stage.

The plot function III

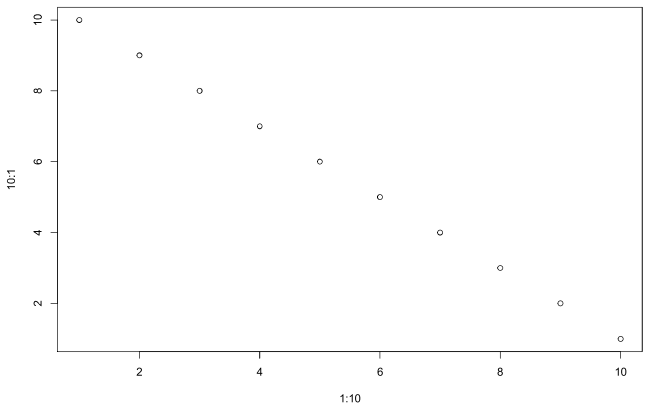
```
plot(x = 1:10)
```



Note the axis labels that are given by default (*Index* and *1:10*, respectively).

The plot function IV

```
plot(x = 1:10, y = 10:1)
```



Note how the axis labels have changed to reflect the input.

Plots in RStudio

In RStudio, plots will be displayed in pane 3. When using the R GUI or the terminal, a new window will be opened. Server behaviour will be different...

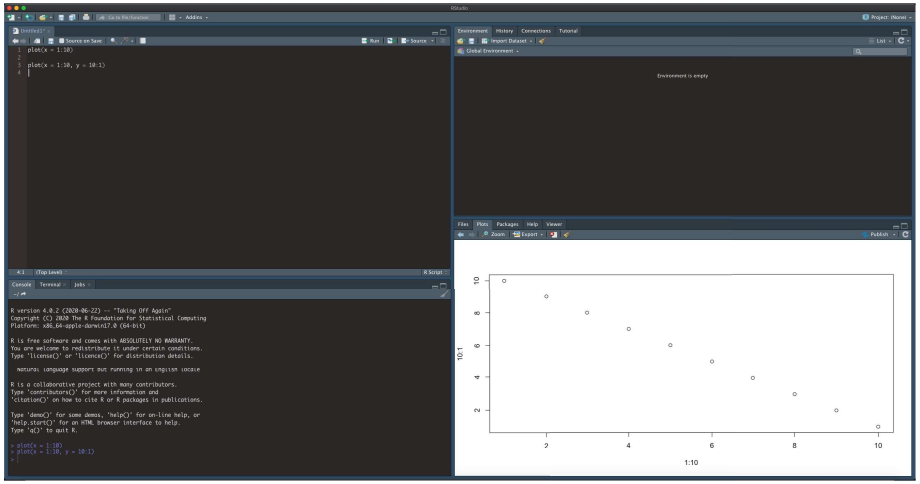
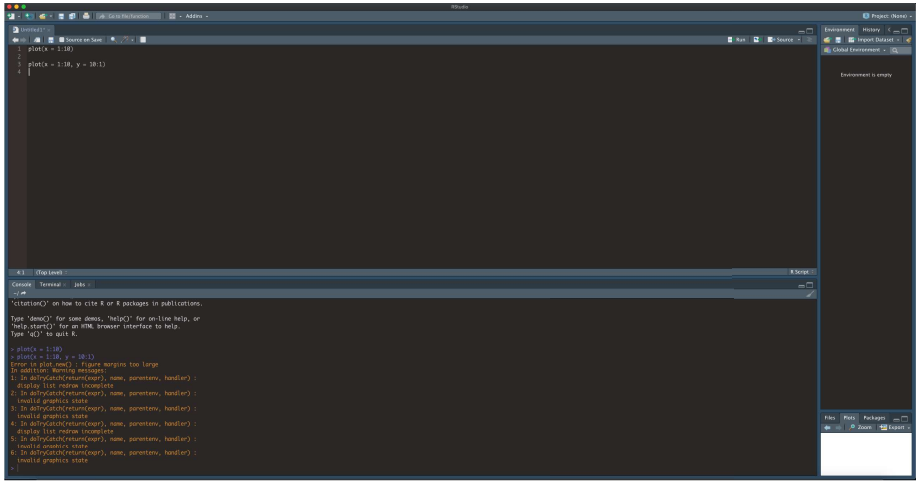


Figure margin problem

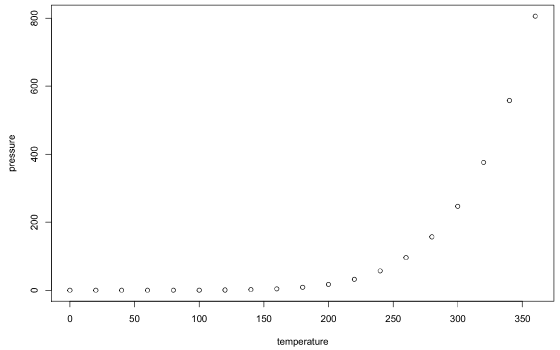
If you encounter an error message stating that the figure margins are too large, then pane 3 is too small (see below). Just resize it and try again.



Using formulas

We can provide the plotting data as a formula as well. Think of the tilde as the equals sign separating the *dependent variable* on the left-hand side and the *independent variable* on the right-hand side. These will be plotted on the y-axis and x-axis, respectively. You then need to set the `data` argument.

```
plot(pressure ~ temperature, data = pressure)
```



Saving plots

We can save plots in different formats like png or pdf, using the respective file extensions as function names (see below). You can tinker with the dimensions and resolution. For printing, a PDF is usually the best option.

After calling these functions, we construct our plot.

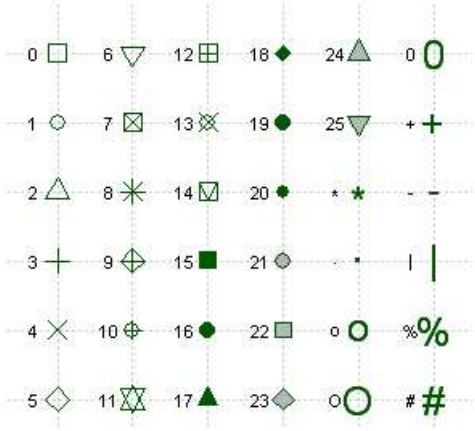
Once the plot is done, the graphics device is closed using `dev.off()`.

ATTENTION:
Note that the plot will not be shown in pane 3 when you plot into a file.

```
png("A_plot.png", width = 1920, height = 1080)
plot(1:10, rep(1, 10))
dev.off()
## pdf
##   2
```


Changing the plotting characters or symbols I

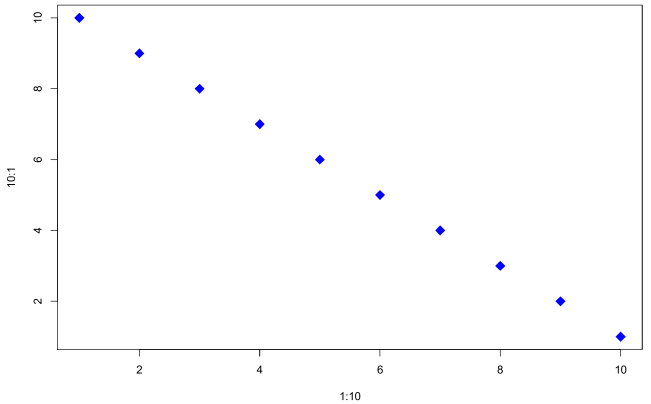
We can easily change the plotting characters or symbols using argument `pch`. The following overview can be found on the *Quick-R* website offered by *DataCamp* and you can reach it here (14/10/2020).



Changing the plotting characters or symbols II

We can also change the colour and the size (the default value being 1) of plotting characters or symbols, using arguments `col` and `cex`, respectively.

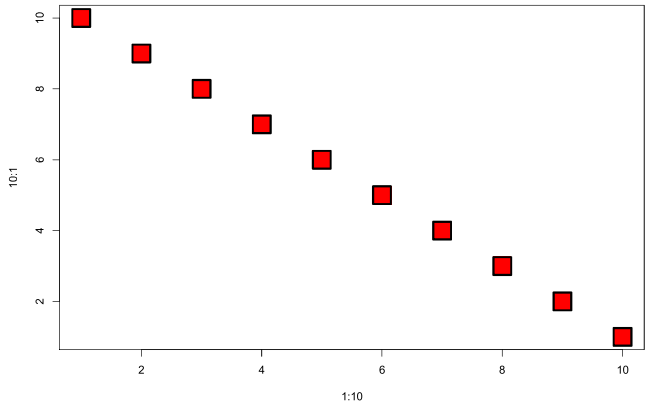
```
plot(x = 1:10, y = 10:1, pch = 18, col = "blue", cex = 2)
```



Changing the plotting characters or symbols III

For some plotting characters or symbols, we can also change the border line width and the background colour, using arguments `lwd` and `bg`, respectively.

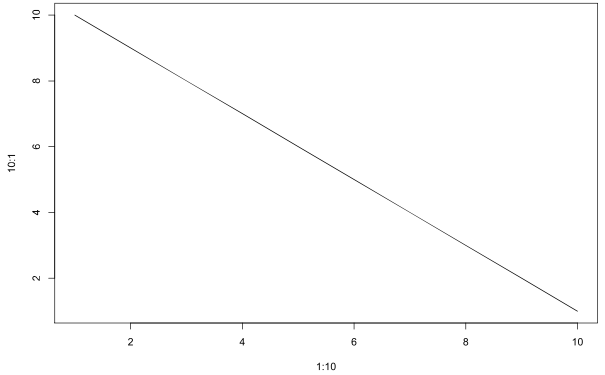
```
plot(1:10, 10:1, pch = 22, cex = 4, lwd = 3, bg = "red")
```



Changing the plot type

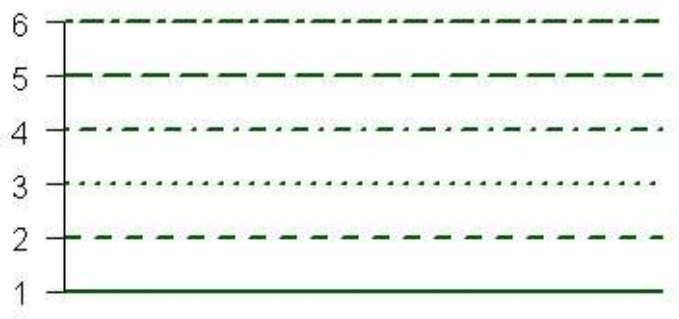
Using the argument `type`, we can define the plot type. We might, e.g., go from points (the default) to lines as follows. For more details, see the documentation for the `plot` function.

```
plot(1:10, 10:1, type = "l")
```



Changing the line type I

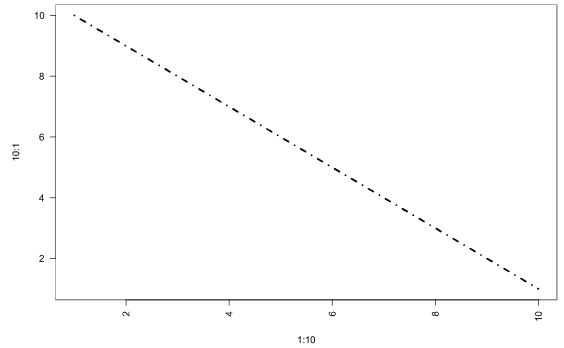
To change the line type in a plot, we use argument `lty`. The following overview can be found on the *Quick-R* website offered by *DataCamp* and you can reach it here (14/10/2020).



Changing the line type II

We can use the argument `lwd` here as well.

```
plot(1:10, 10:1, type = "l", lty = 4, lwd = 3, las = 2)
```



Note that we changed the orientation of the axis labels using argument `las`. For more details, see the documentation for `par`.

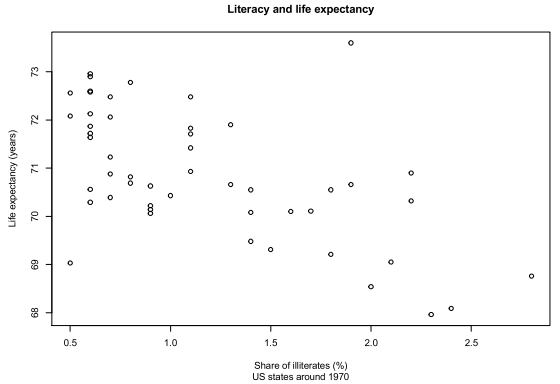
Another data set

We now want to load another data set into our R session. We are specifically dealing with a matrix containing some key statistics for US states around 1970. Our first focus will be the percentage of illiterates in the total population and the life expectancy in years.

```
data(state)
head(state.x77[, c("Illiteracy", "Life Exp")])
##            Illiteracy Life Exp
## Alabama           2.1    69.05
## Alaska            1.5    69.31
## Arizona           1.8    70.55
## Arkansas          1.9    70.66
## California        1.1    71.71
## Colorado          0.7    72.06
ill <- state.x77[, "Illiteracy"]
lif <- state.x77[, "Life Exp"]
```


Adding titles and axis labels

```
plot(ill, lif, main = "Literacy and life expectancy",
     sub = "US states around 1970",
     xlab = "Share of illiterates (%)",
     ylab = "Life expectancy (years)"
     )
```



Identifying points in a scatterplot

Using the function `identify`, you can identify to which state the rather apparent outlier in the scatterplot on the previous slide belongs.

To do so, provide the x- and y-coordinates, `ill` and `lif` in our case, as the first two arguments and the rownames of the original matrix (`state.x77`) as the `labels` argument.

After calling the function, you can click on the outlier in the plot and then hit the ESC key.

Here, a state name should be added to the plot.

Locating points in a scatterplot

If you want to locate points in a scatterplot, you can do so by using the function `locator` without any explicitly set arguments.

Like in the case of `identify`, after calling the function, you can click on the desired points in the plot and then hit the ESC key after you are done.

The output, again printed to the console, will now be the coordinates of the points you clicked on.

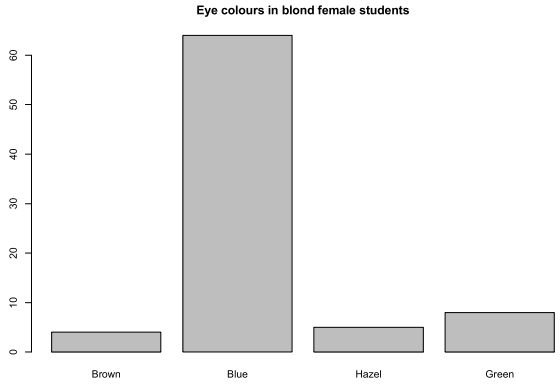
This can be a useful tool to find coordinates to place additional items like text or symbols in your plot after the initial *sketch* is done. This is what we call low-level plotting and we will discuss it in the next subsection.

If you want to save the coordinates for later use just combine the call to `locator` with an assignment. The resulting object will be a list with two elements (`x` and `y`, respectively).

Barplots I

To produce barplots, we use the function `barplot`.

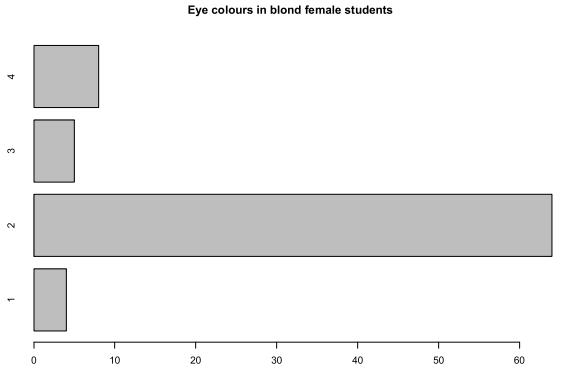
```
blond_females <- HairEyeColor["Blond", , "Female"]
barplot(blond_females,
        main = "Eye colours in blond female students")
```



Barplots II

We can also *rename* the bars and change their orientation.

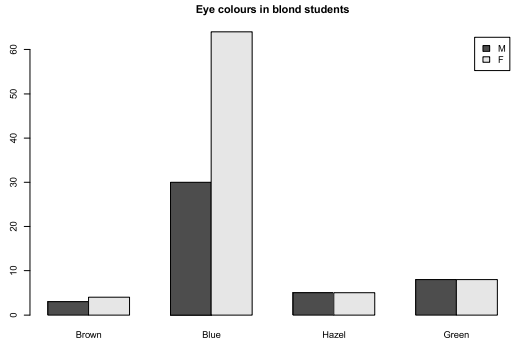
```
barplot(blond_females, horiz = TRUE,
        main = "Eye colours in blond female students",
        names.arg = 1:length(blond_females))
```



Barplots III

Barplots may also include different groups, when the input is a matrix.

```
blond_students <- HairEyeColor["Blond", , ]
barplot(t(blond_students), beside = TRUE, # otherwise stacked
        legend.text = c("M", "F"),
        main = "Eye colours in blond students")
```



apply I

The function `apply` is used to apply other (potentially anonymous) functions (argument `FUN`) per row, per column, or per row and column (argument `MARGIN`) of a matrix or data frame.

```
females <- HairEyeColor[, , "Female"]
(hair <- apply(females, MARGIN = 1, FUN = sum)) # row sums
## Black Brown   Red Blond
##    52   143    37    81
rowSums(females)
## Black Brown   Red Blond
##    52   143    37    81
apply(females, MARGIN = 2, FUN = sum) # column sums
## Brown  Blue Hazel Green
##   122   114    46    31
hair_relative <- hair / sum(hair)
# relative frequencies of hair colours
```

apply II

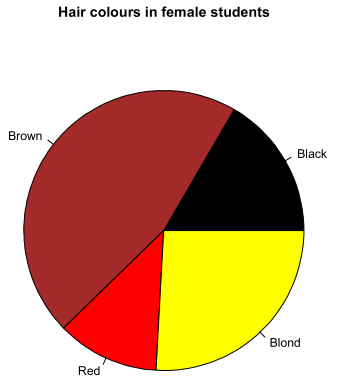
However, it can also be used with an array. Here, we aggregate over the first and fourth dimension of the built-in data set `Titanic`, class and survival status, respectively.

```
apply(Titanic, c(1, 4), sum)
##       Survived
## Class    No Yes
##   1st   122 203
##   2nd   167 118
##   3rd   528 178
##   Crew  673 212
```


Pie charts

Using `pie`, we can plot pie charts.

```
pie(hair_relative, main = "Hair colours in female students",
    col = c("black", "brown", "red", "yellow"))
```



Colours

There are several ways to define colours in R:

1. Index (`numeric`)
2. Name (`character`)
3. Hexadecimal code (`character`) → Not discussed here
4. RGB code (function)

Some packages extend the options regarding colours, like e.g. `RColorBrewer` and `wesanderson`.

For a very useful cheatsheet by Melanie Frazier see link (15/10/2020).

Indexing colours I

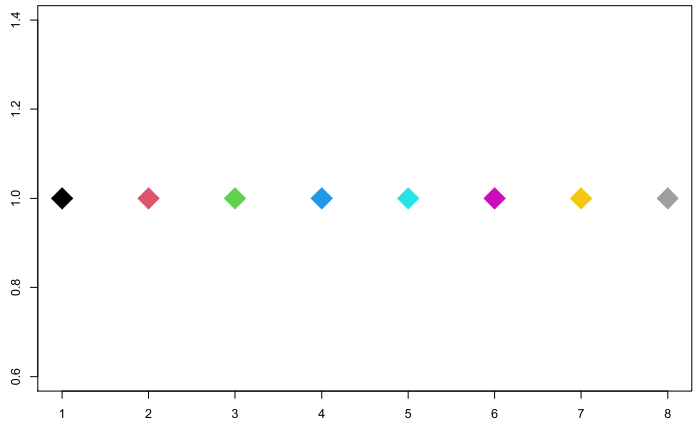
Indexing can either refer to a colour palette R uses in function calls etc. or to all 657 colours built into R.

```
length(palette())
## [1] 8
palette()[1:4]; palette()[5:8]
## [1] "black"   "#DF536B" "#61D04F" "#2297E6"
## [1] "#28E2E5" "#CD0BBC" "#F5C710" "gray62"
head(colours(), 3) # head(colors(), 3)
## [1] "white"         "aliceblue"     "antiquewhite"
colours()[62]
## [1] "cornflowerblue"
```

To see all built-in colours, see the document by Tian Zheng linked here (19/10/2020).

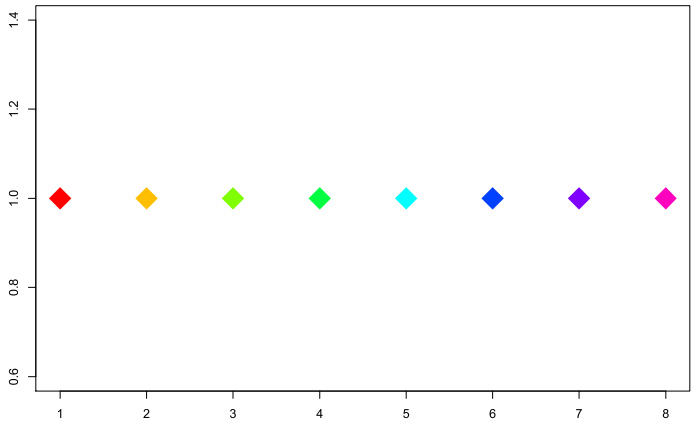
Indexing colours II

```
plot(1:8, rep(1, 8), col = 1:8, pch = 18, cex = 4,
     xlab = "", ylab = "") # no axis labels
```



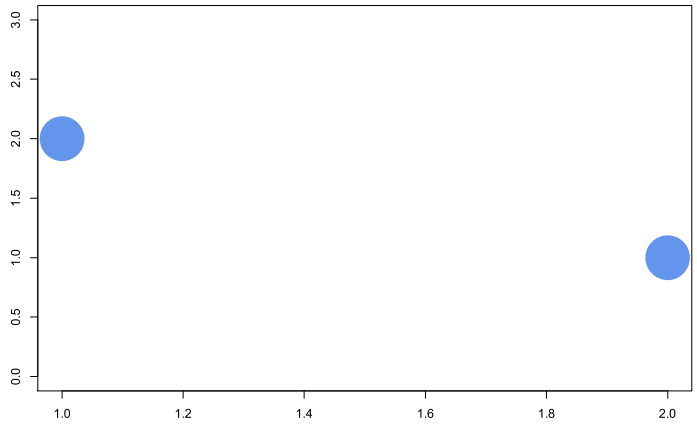
Indexing colours III

```
plot(1:8, rep(1, 8), col = rainbow(8), pch = 18, cex = 4,
     xlab = "", ylab = "")
```



Indexing colours IV

```
plot(1:2, 2:1, col = colours()[62], pch = 16, cex = 8,
     xlab = "", ylab = "", ylim = c(0, 3)) # y-axis limits
```



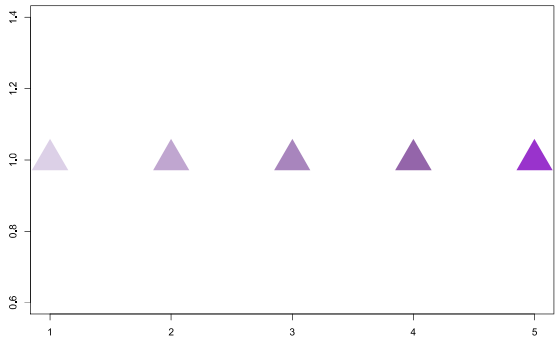
Naming colours

We saw examples of directly naming colours earlier in this chapter. We can just use all 657 colour names as character strings.

Using RGB code I

We can also provide intensities of the **R**ed, **G**reen, and **B**lue channels as well as alpha transparency in order to *mix* our own colours.

```
plot(1:5, rep(1, 5),
     col = rgb(0.6, 0.2, 0.8, alpha = seq(0.2, 1, 0.2)),
     pch = 17, cex = 6, xlab = "", ylab = "")
```



Using RGB code II

You can derive RGB intensities from given colours as well using `col2rgb`. The output is a matrix.

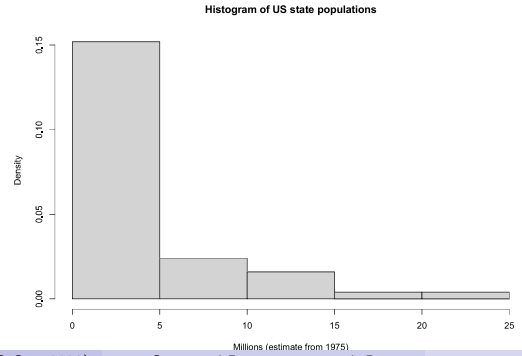
255 is the maximum intensity for a channel.

```
col2rgb("cornflowerblue")
##       [,1]
## red    100
## green  149
## blue   237
```


Histograms I

We can plot histograms using the function `hist`.

```
histogram <- hist(state.x77[, "Population"] / 1000,
                  main = "Histogram of US state populations",
                  xlab = "Millions (estimate from 1975)",
                  freq = FALSE) # areas sum to one
```



Histograms II

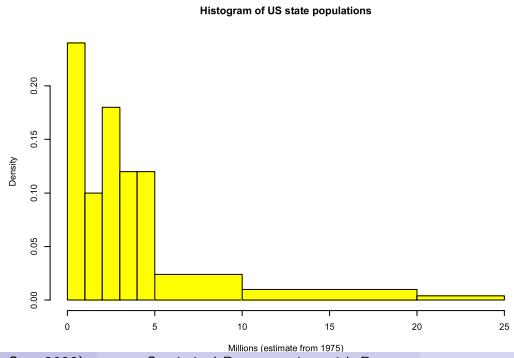
The argument `plot` is set to `TRUE` by default. The function output (a list) contains the bin breaks, the bin counts, etc.

```
str(histogram)
## List of 6
##  $ breaks  : num [1:6] 0 5 10 15 20 25
##  $ counts  : int [1:5] 38 6 4 1 1
##  $ density : num [1:5] 0.152 0.024 0.016 0.004 0.004
##  $ mids    : num [1:5] 2.5 7.5 12.5 17.5 22.5
##  $ xname   : chr "state.x77[, \"Population\"]/1000"
##  $ equidist: logi TRUE
##  - attr(*, "class")= chr "histogram"
breaks <- histogram$breaks
density <- histogram$density
sum((breaks[-1] - breaks[-length(breaks)]) * density)
## [1] 1
```

Histograms III

We can also define the breaks and change the bin colour.

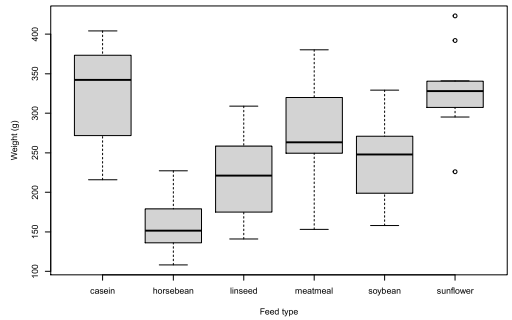
```
hist(state.x77[, "Population"] / 1000, col = "yellow",
     main = "Histogram of US state populations",
     xlab = "Millions (estimate from 1975)",
     freq = FALSE, breaks = c(0, 1:5, 10, 20, 25))
```



Box plots

To produce box plots, we use the function `boxplot`. In our example, we use the built-in data set `chickwts`, containing chicken weights and feed types.

```
boxplot(chickwts$weight ~ chickwts$feed,
        xlab = "Feed type", ylab = "Weight (g)")
```

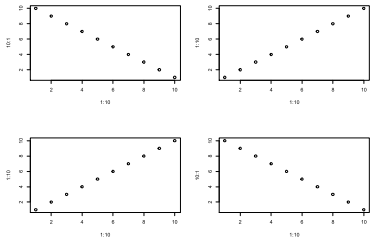


A box is drawn to visualise the weight distribution for every feed type.

Multiple plots in one figure

We might want to draw multiple plots within one figure, e.g. to compare data. To this end, we can use the function to set/query graphical parameters `par`.

```
par(mfrow = c(2, 2)) # 2 rows, 2 columns, by rows
plot(1:10, 10:1); plot(1:10, 1:10)
plot(1:10, 1:10); plot(1:10, 10:1)
```

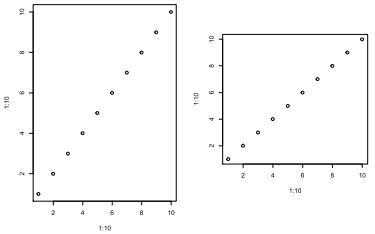


```
par(mfrow = c(1, 1)) # reset once done
```


Setting the margins I

You can change the margins of a plot from their default values of 5.1, 4.1, 4.1, and 2.1 lines for the bottom, left, top, and right margin, respectively. The figure region contains both, the plot region and the margins.

```
par(mfrow = c(1, 2))
plot(1:10, 1:10)
par(mar = c(10, 4, 8, 2)); plot(1:10, 1:10)
```



```
par(mfrow = c(1, 1))
```

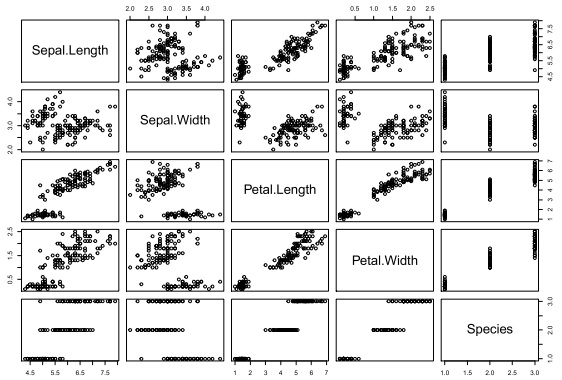
Setting the margins II

We could also set the outer margins using the argument `oma` to `par`. The outer margins are the space still left by the figure region on the device region. However, we will not discuss this here.

Multivariate data overview

When visualising multivariate data sets like the one on different species of iris in the built-in data set `iris`, we see the generic nature of the function `plot` again.

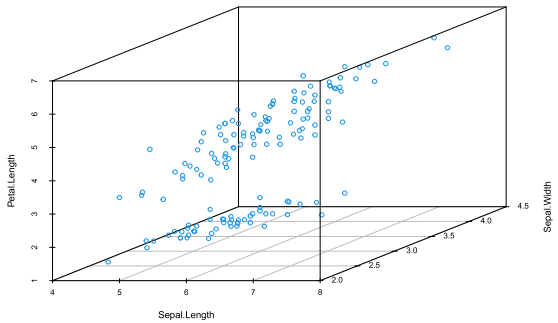
```
plot(iris)
```



Simple 3D scatterplots

We can use the function `scatterplot3d` of the package of the same name to draw simple 3D scatterplots.

```
# install.packages("scatterplot3d")
library(scatterplot3d)
scatterplot3d(iris[, 1:3], color = 4)
```



A shortcoming of base graphics

We can already see that the visualisation of multivariate data seems to be a bit of a shortcoming of base graphics. The next sections hint at ways to overcome this.

Adding lines and plotting symbols I

We want to add some information to a scatterplot of the savings ratio and the share of the population that is less than 15 years old for 50 countries, averaged for 1960–1970. The data are in the data set `LifeCycleSavings`.

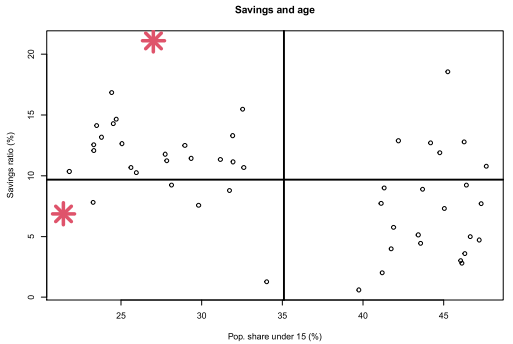
We first add a horizontal line at the savings ratio average and a vertical line at the population share average to the plot.

We then highlight the data point with the highest savings ratio and the one with the lowest population share by adding another plotting symbol to them.

ATTENTION:
You can only use low-level plotting when a plot was *started* beforehand. Modifications cannot be undone. If you would like to change them, you have to start anew with a call of the original plotting function.

Adding lines and plotting symbols II

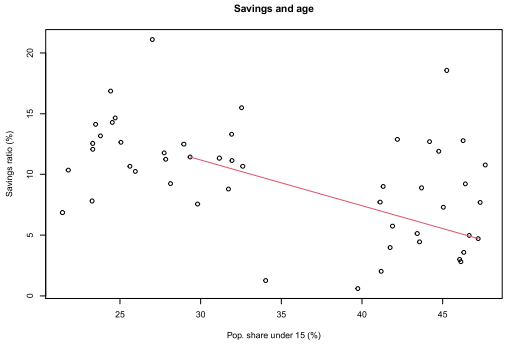
```
dat <- LifeCycleSavings[, c("pop15", "sr")]
plot(dat$pop15, dat$sr, main = "Savings and age",
 xlab = "Pop. share under 15 (%)", ylab = "Savings ratio (%)")
abline(h = mean(dat$sr), v = mean(dat$pop15), lwd = 3)
points(dat[c(which.max(dat$sr), which.min(dat$pop15)), ],
       pch = 8, cex = 4, lwd = 6, col = 2)
```



Adding lines and plotting symbols III

We could also connect any two points with a line (here: first and last entry).

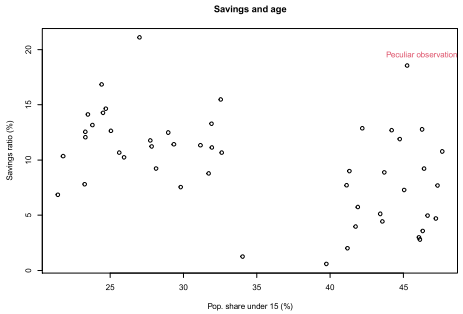
```
plot(dat$pop15, dat$sr, main = "Savings and age",
 xlab = "Pop. share under 15 (%)", ylab = "Savings ratio (%)")
lines(x = dat[c(1, nrow(dat)), 1],
      y = dat[c(1, nrow(dat)), 2], col = 2)
```



Adding text

We now add text to a peculiar data point.

```
top_right <- dat[, 1]>mean(dat[, 1]) & dat[, 2]>mean(dat[, 2])
temp<-dat[top_right, ]; peculiar<-temp[which.max(temp$sr), ]
plot(dat$pop15, dat$sr, main = "Savings and age",
 xlab = "Pop. share under 15 (%)", ylab = "Savings ratio (%)")
text(peculiar + 1, labels = "Peculiar observation", col = 2)
```



History

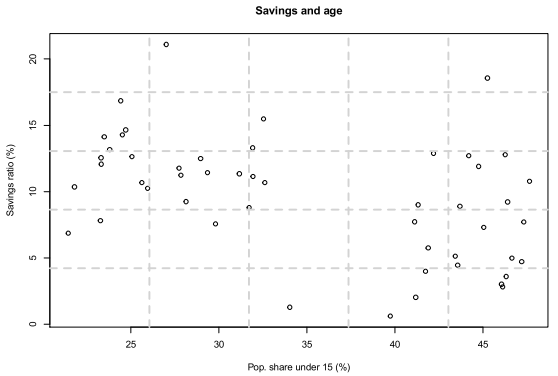
As you saw on the previous slides, we just re-used the original high-level plotting call of the scatterplot.

If we only want to try out low-level plotting variations, we can just call `plot` with all the arguments we used before by **clicking into the console** and browsing the (command) history using the **up arrow and down arrow keys**. Once the needed command appears in the console, we just hit return.

Alternatively, we could go to the *History* tab in pane 4 and click the command we need, once again hitting return after it appears in the console.

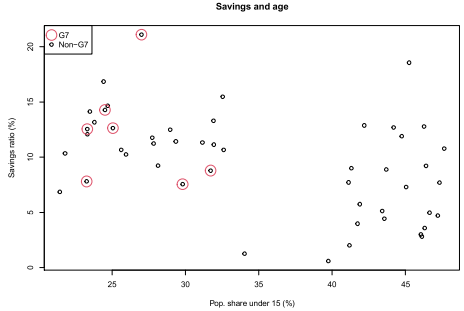
Adding a grid

```
plot(dat$pop15, dat$sr, main = "Savings and age",
 xlab = "Pop. share under 15 (%)", ylab = "Savings ratio (%)")
grid(nx = 5, ny = 5, lwd = 3)
```



Adding legends I

```
g7 <- c("United States", "Japan", "Germany", "France",
 "United Kingdom", "Italy", "Canada")
plot(dat$pop15, dat$sr, main = "Savings and age",
 xlab = "Pop. share under 15 (%)", ylab = "Savings ratio (%)")
points(dat[rownames(dat) %in% g7, ], col = 2, cex = 3)
legend("topleft", legend = c("G7", "Non-G7"), pch = 1,
       col = 2:1, pt.cex = c(3, 1))
```



Adding legends II

On the last slide, we used the character string `"topleft"` as the first argument to position the legend in the top-left corner of the plotting area. To position it in the bottom-right corner instead, we would use `"bottomright"` etc.

However, we can also directly specify x- and y-coordinates using the arguments `x` and `y`, respectively.

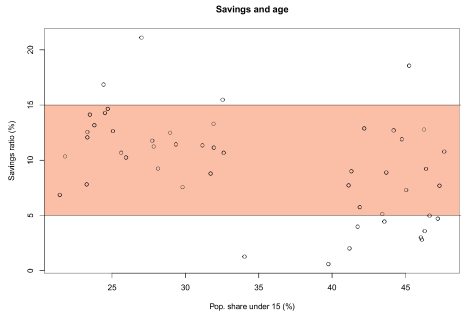
Setting the argument `bty` to `"n"` leads to an omission of the box around the legend.

There are many more arguments to `legend` that can be used to fine-tune plotting symbols etc.

Adding rectangles

To highlight certain areas, you can use rectangles (function `rect`).

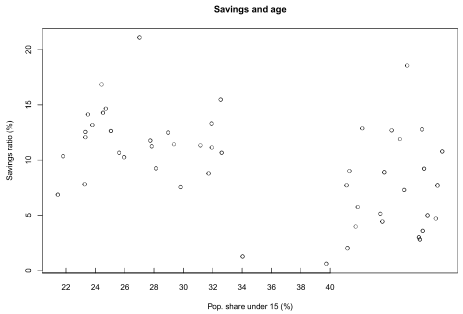
```
plot(dat$pop15, dat$sr, main = "Savings and age",
 xlab = "Pop. share under 15 (%)", ylab = "Savings ratio (%)")
rect(xleft = 0.9 * min(dat$pop15), ybottom = 5,
     xright = 1.1 * max(dat$pop15), ytop = 15,
     col = adjustcolor("red", alpha.f = 0.3))
```



Modifying an axis

If we do not want to use a default axis, which tries to match the data, we can suppress it in high-level plotting (argument `xaxt`) and add it afterwards.

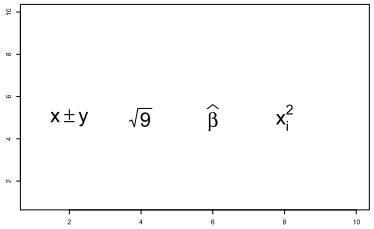
```
plot(dat$pop15, dat$sr, main = "Savings and age", xaxt = "n",
 xlab = "Pop. share under 15 (%)", ylab = "Savings ratio (%)")
axis(side = 1, at = seq(20, 40, 2),
     labels = seq(20, 40, 2), tcl = 0.5)
```



Mathematical annotation

We can use mathematical annotation in plot components, like texts, axis labels, and legends, as well. See `?plotmath` for details. Note that we draw an empty plot first below.

```
plot(1:10, 1:10, type = "n", xlab = "", ylab = "")
text(2, 5, expression(x %+-% y), cex = 3)
text(4, 5, expression(sqrt(9)), cex = 3)
text(6, 5, expression(widehat(beta)), cex = 3)
text(8, 5, expression(x[i]^{2}), cex = 3)
```



The strength of `lattice`

This section draws from lecture slides of a course given by Ross Ihaka that are available here (20/10/2020).

The `lattice` package, contributed by Deepayan Sarkar, does not rely on the code of the base graphics system as found in the package `graphics`. Instead, it uses the code framework called *grid*, developed by Paul Murrell.

ATTENTION:
You cannot use functions of one system within the other.

`par` does not work for `lattice` plots.

`lattice` makes the production of figures with multiple *regions*, which are quite useful for the visualisation of multivariate data (of mixed scales), much easier.

`lattice` therefore facilitates the production of *trellis* or *faceted graphs*. We will see what this means shortly. In the meantime, try `demo(lattice)`.

Conditioning as core idea

The core idea behind trellis graphs is that we produce a set of plots of the same type **conditioned** on values of single or cross-combinations of multiple variables, respectively.

The conditioning variable will typically be a kind of factor derived either from categories of non-metric variables or intervals of metric variables, respectively.

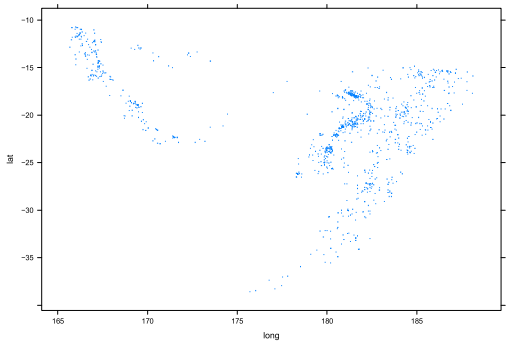
We now load the `lattice` package.

```
#install.packages("lattice")
library(lattice)
```


Simple scatterplot

We now first produce a simple scatterplot of the locations of earthquakes off Fiji starting from 1964, given in the built-in data set `quakes`. However, we do not use the `graphics` package but rather the `lattice` package, using the function `xyplot`. The *readability* is improved.

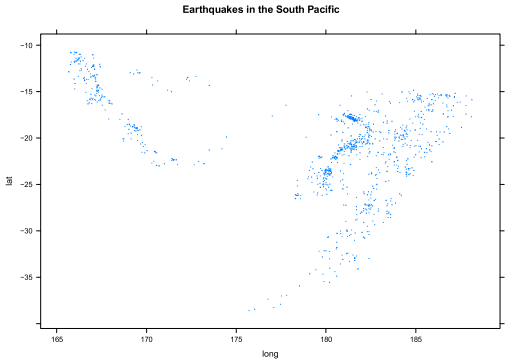
```
xyplot(lat ~ long, data = quakes, pch = ".")
```



Updating existing plots

A neat feature of `lattice` is that we can update plots that already *exist* and are saved as lists.

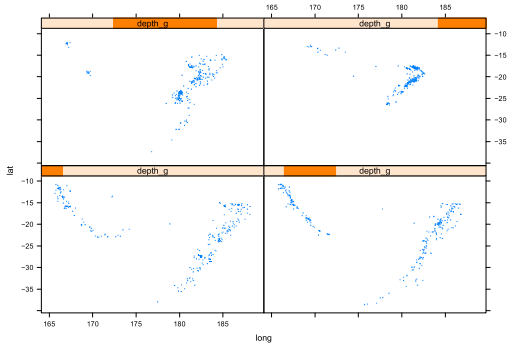
```
scatter <- xyplot(lat ~ long, data = quakes, pch = ".")
update(scatter, main = "Earthquakes in the South Pacific")
```



Conditioning on one variable

We now want to condition on a newly constructed depth group indicator. The new indicator is a `shingle`, which is a generalisation of factors to continuous variables. The conditioning variable is given after the `|`.

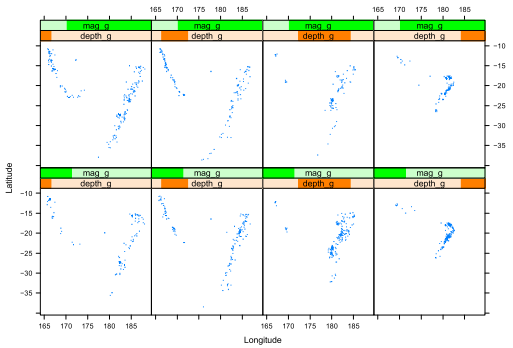
```
depth_g <- equal.count(quakes$depth, number = 4, overlap = 0)
xyplot(lat ~ long | depth_g, data = quakes, pch = ".")
```



Conditioning on two variables

Note the *reading direction* of the plot on the previous slide.
We now want to condition on two variables.

```
mag_g <- equal.count(quakes$mag, number = 2, overlap = 0)
xyplot(lat ~ long | depth_g * mag_g, data = quakes, pch = ".",
       xlab = "Longitude", ylab = "Latitude")
```



Other plot types

We just worked with scatterplots. But we can produce many other plot types with `lattice` as well (including boxplots, 3D wireframes, etc.). For an overview, see this link (20/10/2020).

Grammar of graphics

The package `ggplot2`, originally developed by Hadley Wickham, builds on the **grammar of graphics** as described in *Graph-Theoretic Scagnostics* by Wilkinson, Anand, and Grossman, which you can find here (20/10/2020), and the 2005 book *The Grammar of Graphics* by Wilkinson. This grammar aims at a concise description, and thereby understanding, of the components of graphics, which in turn facilitates the construction of useful graphics. See also Wickham, H. (2010): A Layered Grammar of Graphics. Journal of Computational and Graphical Statistics, 19(1), pp. 3–28.

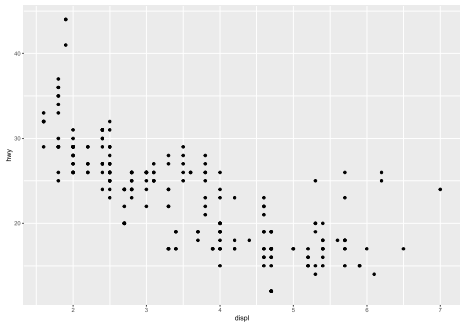
`ggplot2` is part of the *tidyverse*, wich collects data science-focussed packages that share many features. See also here (20/10/2020).

This section leans on chapter 3 of the book *R for Data Science* by Garrett Grolemund and Hadley Wickham, which is free to use here (20/10/2020).

Basic ggplot I

We visualise the relationship between engine size and fuel efficiency in cars.

```
#install.packages("ggplot2")
library(ggplot2)
ggplot(data = mpg) + # The + has to be at the end of a line.
  geom_point(mapping = aes(x = displ, y = hwy))
```



Basic ggplot II

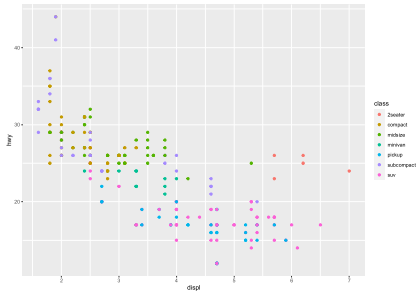
A basic ggplot has two components:

1. **Call to `ggplot`**
   Defines data set to be used and creates coordinate system
2. **Addition of *layers* (at least one)**
   Here: Layer of points → scatterplot

Basic ggplot III

By mapping the variable `displ` (engine size) to the x-axis and the variable `hwy` (miles per gallon) to the y-axis, respectively, we defined aesthetics of the plot. We can now map a third variable to an aesthetic, namely the type of car (variable `class`). With `graphics`, this would be much more work.

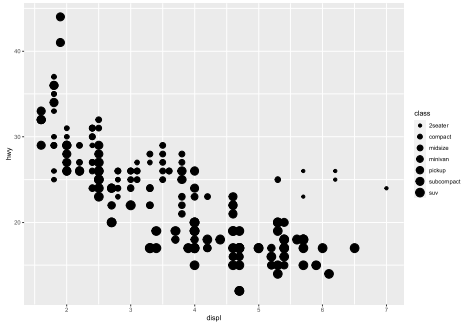
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x=displ, y=hwy, colour=class))
```



Basic ggplot IV

`ggplot2` also gives you some hints.

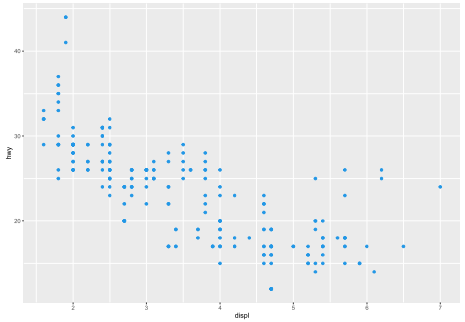
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x=displ, y=hwy, size=class))
## Warning: Using size for a discrete variable is not advised.
```



Basic ggplot V

We can also actively set properties of our geoms or geometric objects.

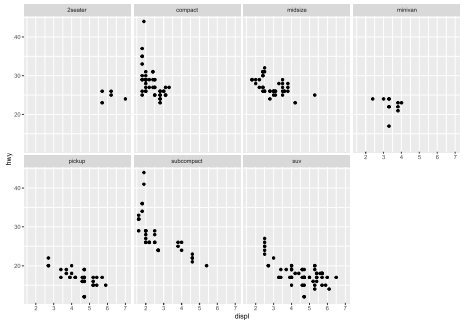
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x=displ, y=hwy), colour = 4)
```



Faceted graphs I

Like with `lattice`, it is rather easy to produce faceted graphs with ggplot2. Here, we use one conditioning variable and `facet_wrap`.

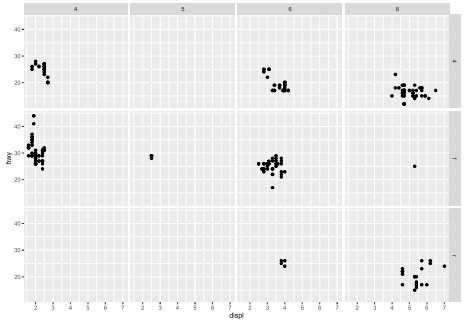
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2) # additional layer
```



Faceted graphs II

Now, we condition on two variables, using `facet_grid`.

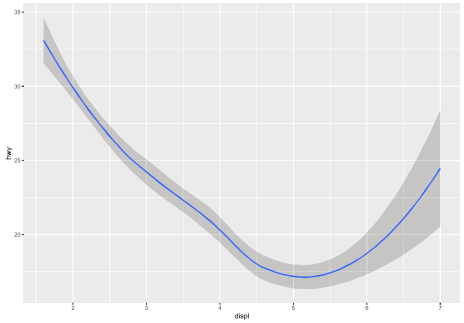
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ cyl) # additional layer
```



Changing geoms

We can easily change the geom, e.g. moving from a scatterplot to a smoothed line fitted to the data.

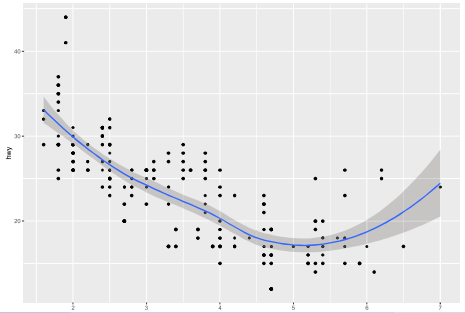
```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Combining geoms I

Or we can combine geoms and globally define mappings for all geoms in the call to ggplot.

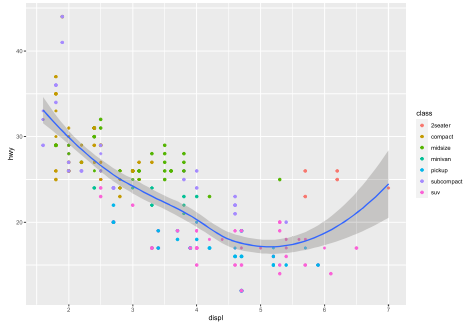
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Combining geoms II

If needed, global mappings can be overwritten within the layers.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(colour = class)) +
  geom_smooth()
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Statistical Programming with R

Chapter 6 - Control structures and functions

Florian Ertz, Jan Pablo Burgard, and Ralf Münnich

Economic and Social Statistics
Economics / Faculty IV
Trier University

Winter semester 2021/2022

Copyright statement

Control structures

You will usually need control structures when you program your own functions or, e.g., complex simulation setups.

Simply put, they will determine how some input is processed along your code.

See also `?Control`.

if I

To introduce a condition into a program, we can use an `if` statement. Only if the condition, formulated as a logical expression within parentheses, is met, a command or several commands is/are executed.

```
some_numbers <- c("3", "2", "6", "1", "8")
if (is.character(some_numbers)) {
  some_numbers <- as.numeric(some_numbers) # silent
}
sum(some_numbers)
## [1] 20
```

Note that we would not need the curly braces enclosing the expression to be executed here. We could have just written it after the parentheses in the same line. However, when we use an expression covering several commands and/or several lines, we need curly braces. They also enhance readability.

if II

```
if (is.character(some_numbers)) print("Oh, characters...")
```


if and else

If we want to *choose* one of two alternatives depending on whether the condition is met or not, we can combine `if` and `else`.

```
if (sum(some_numbers) > 10) {
  print("The sum is reasonably large.")
} else {
    print("The sum is rather small.")
  }
## [1] "The sum is reasonably large."
```

Note that we also use curly braces for the `else` expression.

ifelse

The function `ifelse` is basically a vectorised variant of the combination of `if` and `else`. It is suitable for simple conditions and simple tasks like *relabeling* vectors etc.

```
ifelse(some_numbers > 5, ">5", "<=5")
## [1] "<=5" "<=5" ">5"  "<=5" ">5"
```

Note that the output is of the same length as the logical vector that is *constructed* by the condition.

Loops

Loops are used to repeat a certain task or certain tasks.
There are two types of loops in R:

1. **`for` loops**
   Repeat task(s) a pre-defined number of times
   Example: Eat numbered muffins 1 through 3.
2. **`while` loops**
   Repeat task(s) as long as condition is met
   Example: While hungry, eat muffins.

for loops I

A `for` loop executes an expression or a group of expressions once for every element of a vector that is provided within parentheses. The loop index, conventionally denoted by an `i`, takes on the first, second, etc. value of the vector in the first, second, etc. repetition of the loop. We can use the loop index itself, e.g. to indicate the progress of the loop in the console.

```
for (i in 1:5) {
  print(i * 3) # 1 * 3, 2 * 3, ...
}
## [1] 3
## [1] 6
## [1] 9
## [1] 12
## [1] 15
```


for loops II

Using `cat`, we can display output in the console. A `for` loop on its own is silent. This is why we used `print` on the previous slide.

```
for (i in 1:5) {
  cat(i, "times 3 is", i * 3, ".\n")
}
## 1 times 3 is 3 .
## 2 times 3 is 6 .
## 3 times 3 is 9 .
## 4 times 3 is 12 .
## 5 times 3 is 15 .
```

Note that the escape sequence `\n` is used to start a new line.

for loops III

```
for (name in c("Hanna", "Julia", "Anna")) {
  cat(name, "loves R!\n")
}
## Hanna loves R!
## Julia loves R!
## Anna loves R!
```


Efficiency of loops I

We could compute the sum of the natural numbers up to a given number using a for loop. Let us time this with system.time as well.

```
system.time( # uses difference between two calls of proc.time
  {
  cum_sum <- 0
  for (i in 1:1e4) {
    cum_sum <- cum_sum + i
  }
  }
)
##    user  system elapsed
##   0.002   0.000   0.002
cum_sum
## [1] 50005000
```


Efficiency of loops II

Now let us use cumsum instead.

```
system.time(
  {
  nat_numbers <- 1:1e4
  cum_sum2 <- cumsum(nat_numbers)[length(nat_numbers)]
  }
)
##    user  system elapsed
##       0       0       0
cum_sum2
## [1] 50005000
```


Efficiency of loops III

And now let us first replicate some data for US states and then do some computations. First using a loop...

```
state_df <- data.frame(rep(state.x77[, "Population"], 1e4),
                       rep(state.x77[, "Area"], 1e4))
pop_area <- numeric(nrow(state_df))
system.time(
  for (i in 1:nrow(state_df)) {
    pop_area[i] <- state_df[i, 1] / state_df[i, 2]
  }
)
##    user  system elapsed
##   8.330   0.033   8.371
```

Efficiency of loops IV

... and then using apply.

```
system.time(apply(state_df, 1, function(x) x[1] / x[2]))
##    user  system elapsed
##   2.469   0.041   2.511
```

ATTENTION:
Avoid loops wherever you can. Using lists and the related functions of the *apply family* is encouraged.

The R Inferno by Patrick Burns is suggested reading for advanced *useRs*. You can find it here (27/10/2020).

Efficiency of loops V

If you have to use loops, you should initialise objects that are changed in the loop before starting it (see slide before the previous one). *Growing* objects in the loop is inefficient. Also try to avoid redundant computations.

```
n <- 5
constant <- 2 * n * pi
some_product <- numeric(n)
for (i in 1:n) {
  some_product[i] <-
    # Bad: 2 * n * pi * (i + 10)
    constant * (i + 10) # Good
}
some_product
## [1] 345.5752 376.9911 408.4070 439.8230 471.2389
```


Efficiency of loops VI

The following example is taken from the 2008 book *Programmieren mit R* by Uwe Ligges.

```
fun1 <- function(n) {
  a <- NULL
  for (i in 1:n) {
    a <- c(a, i^2)
  }
}
fun2 <- function(n) {
  a <- numeric(n)
  for (i in 1:n) {
    a[i] <- i^2
  }
}
fun3 <- function(n) a <- (1:n)^2
```


Efficiency of loops VII

```
system.time(fun1(2e5))
##    user  system elapsed
##  50.923  22.315  73.327
system.time(fun2(2e5))
##    user  system elapsed
##   0.012   0.000   0.012
system.time(fun3(2e5))
##    user  system elapsed
##   0.001   0.000   0.001
# Do not run with any of the other functions
system.time(fun3(2e8))
##    user  system elapsed
##   0.912   0.489   1.401
```

Efficiency of loops VIII

One might be tempted to use loops when analysing subsets of data sets. The built-in data set `OrchardSprays` contains 8 groups determined by the treatment level. We want to average the responses within these groups. Pasting and changing a line of code seven times is not an option.

```
treatment <- levels(OrchardSprays$treatment)
avg_response <- numeric(length(treatment))
names(avg_response) <- treatment
for (i in treatment) {
  members <- which(OrchardSprays$treatment == i)
  avg_response[i] <- mean(OrchardSprays$decrease[members])
}
avg_response
##      A      B      C      D      E      F      G      H
##  4.625  7.625 25.250 35.000 63.125 69.000 68.500 90.250
```


tapply I

To analyse subsets of data sets, you should try to use `tapply`.

Note that the second argument, which defines the groups, can either be a single factor or a list of several factors.

```
tapply(OrchardSprays$decrease, OrchardSprays$treatment, mean)
##      A      B      C      D      E      F      G      H
##  4.625  7.625 25.250 35.000 63.125 69.000 68.500 90.250
```

Besides saving many key strokes as compared to the loop solution of the previous slide, `tapply` is also much faster, which is more obvious with more intensive tasks than this one.

tapply II

```
ranges <-
tapply(OrchardSprays$decrease, OrchardSprays$treatment, range)
ranges[1:4]
## $A
## [1]  2 12
##
## $B
## [1]  4 14
##
## $C
## [1]  9 84
##
## $D
## [1] 20 57
```


tapply III

```
# Built-in data set mtcars
tapply(mtcars$mpg,
       list(mtcars$am, mtcars$cyl),
       mean
       )
##        4        6     8
## 0 22.900 19.12500 15.05
## 1 28.075 20.56667 15.40
```

break and next

If needed, we can also break a loop or jump to the next iteration.

```
for (i in 1:4) {
  if (i == 3) break
  print(i)
}
## [1] 1
## [1] 2
for (i in 1:4) {
  if (i == 3) next
  print(i)
}
## [1] 1
## [1] 2
## [1] 4
```


while loops I

A while loop executes an expression or a group of expressions as *long* as a condition is met. Here, we do not know the number of repetitions in advance. Once the condition is not met anymore, the loop is stopped.

Let us approximate Euler's number.

```
i <- 0 # Repetition counter
e <- exp(1) # Target value
approx_e <- 0 # Initial approximation
while (e - approx_e > 10^(-5)) {
  approx_e <- approx_e + 1 / factorial(i)
  i <- i + 1 # Counter has to be updated "manually"
}
i; approx_e; e
## [1] 9
## [1] 2.718279
## [1] 2.718282
```


Infinite loops

ATTENTION:
Avoid infinite loops like the one below.

```
# Do not run: lala <- 3; while (lala > 2) lala <- lala + 1
```

To **stop the execution of a currently running command**, either hit the **escape key** or click the **stop sign button** on the upper right-hand corner of the console.

Using the **escape key**, you can also **stop the input of a command** and go back to the prompt in the console.

Programming functions

As already mentioned in passing in chapter 3, we can program our own functions to deal with recurring and usually more complex tasks. To re-use such functions, we have to give them a name. So these functions are not *anonymous* anymore.

We are free to use as many arguments as we want.

Let us start with a very straightforward example.

```
my.sum <- function(summand_one, summand_two) { # funct. head
  summand_one + summand_two # funct. body
}
my.sum(9, 1)
## [1] 10
```

Operators

By the way and as previously noted, operators are *hidden* function calls as well. Below, backticks just tell R to use a character as a name.

```
`*`(3, 3)
## [1] 9
`>`(10, 5)
## [1] TRUE
`[`(seq(3, by = 3, length.out = 5), 3)
## [1] 9
```


Looking into functions

To look into the code of functions, just type their name, without parentheses, into the console and hit return.

```
my.sum
## function(summand_one, summand_two) { # funct. head
##   summand_one + summand_two # funct. body
## }
```


Default argument values

We can set default argument values as well. Here, the second summand is 1 by default, i.e. if it is not explicitly changed in the function call.

```
my.sum <- function(summand_one, summand_two = 1) {
  summand_one + summand_two
}
my.sum(20)
## [1] 21
```


Defining function output I

If we wish to return more information to the user, we can do so.

```
my.sum <- function(summand_one, summand_two = 1) {
  result <- summand_one + summand_two
  output <- c(summand_one, summand_two, result)
  names(output) <- c("Summand1", "Summand2", "Sum")
  output
}
my.sum(20)
## Summand1 Summand2      Sum
##       20        1       21
```

Defining function output II

Many functions use lists as their output.

```
my.stats <- function(x, y) {
  x_range <- range(x)
  x_mean <- mean(x)
  x_in_y <- any(x %in% y)
  report <- paste("We checked ", length(x),
                  " units. ", ifelse(x_in_y, "Some ", "No "),
                  "peculiar values were found.", sep = ""
                  ) # more on paste shortly
  output <- list(Range = x_range, Mean = x_mean,
                 Check = x_in_y, Report = report
                 )
  output
}
```


Defining function output III

We can now assign the function output to a new object to re-use it.

```
fun_out <- my.stats(y = 10:12, x = c(5, 10, 7, 2, 21))
fun_out
## $Range
## [1]  2 21
##
## $Mean
## [1] 9
##
## $Check
## [1] TRUE
##
## $Report
## [1] "We checked 5 units. Some peculiar values were found."
```


paste I

The arguments of paste are first converted to character strings, if needed, and then concatenated. The string given by the argument sep separates the individual arguments in the resulting character string. Data source: Trier University (28/10/2020).

```
students <- c(2850, 2284, 1120, 2830, 1993, 1091)
text_part <- "Faculty"; pasted <-
paste(text_part, 1:length(students), ":", students, sep = " ")
as.matrix(pasted)
##      [,1]
## [1,] "Faculty 1 : 2850"
## [2,] "Faculty 2 : 2284"
## [3,] "Faculty 3 : 1120"
## [4,] "Faculty 4 : 2830"
## [5,] "Faculty 5 : 1993"
## [6,] "Faculty 6 : 1091"
```


paste II

paste is really useful in the context of visualisation and when working with, e.g., survey data (think the construction of unique IDs).

While the argument sep defines the *separator* for the corresponding elements, the argument collapse defines a separator for the individual results, leading to a single character string as the result.

```
some_names <- c("John", "Paul", "George", "Ringo")
paste(some_names, collapse = " and ")
## [1] "John and Paul and George and Ringo"
```

return I

Using `return`, we can save time by preventing unnecessary processing.

```
locate.peculiarities <- function(x, y) {
  x_in_y <- any(x %in% y)
  if (!x_in_y) {
    return("No peculiarities were found.")
  } else {
      which(x %in% y)
    }
}
```


return II

```
locate.peculiarities(x = c(3, 1, 13, 14, 6), y = 10:12)
## [1] "No peculiarities were found."
locate.peculiarities(x = c(3, 1, 10, 12, 6), y = 10:12)
## [1] 3 4
```

ATTENTION:
The example above served only to demonstrate the concept. In practice, you should avoid function output with changing data types.

Three dots argument I

We can pass additional arguments that are not explicitly defined in advance to functions in the function body as well. This is done using the *three dots argument*.

```
iphone.sales <- function(x, ...) {
      new_number <- round(x, ...)
      paste("Apple sold approx. ", new_number,
            " bn USD in iPhones.", sep = ""
            )
}
```


Three dots argument II

```
iphone.sales(26.418)
## [1] "Apple sold approx. 26 bn USD in iPhones."
iphone.sales(26.418, digits = 1)
## [1] "Apple sold approx. 26.4 bn USD in iPhones."
```

Warnings

To notify the users of your function of peculiar inputs, unlikely results etc. that are **possible but very unlikely**, you can build warnings into your functions. If such a warning is triggered, the user will be notified in the console but the **function will still be executed**.

```
my.mean <- function(x) {
  if (any(x < 10)) {
    warning("Some observations are unusually small.")
  }
  mean(x)
}
my.mean(9:30)
## Warning in my.mean(9:30): Some observations are unusually s
## [1] 19.5
```


Errors

Errors should only be triggered if **things really go awry**. **Function execution will be stopped** after an error has been triggered.

```
hello <- function(x) {
  if (!is.character(x)) {
    stop(paste0(x, " is not a name!"))
  }
  paste0("Hello, ", x, "!")
}
hello("Florian") # Note how paste0 works.
## [1] "Hello, Florian!"
hello(11)
## Error in hello(11): 11 is not a name!
```


`source`

If you need a selection of functions in your current project, it might be worthwhile to save them in a separate R script and then source this whole script at the beginning of your current work script. You do so by calling `source` and passing the path to the *function script* as a character string as the argument.

Writing your own R package

Once you have written a bunch of thematically related functions, you might either want a bit of organisation for yourself or to contribute to the R community. You should then think about writing your own R package. Although we do not have enough time to cover the topic here, you are encouraged to check out the following resources that offer nice introductions.

- Karl Broman:
  R package primer - a minimal tutorial
  → Link (28/10/2020)
- Hadley Wickham and Jennifer Bryan:
  R Packages - Organize, Test, Document and Share Your Code
  → Link (28/10/2020)
- Hilary Parker:
  Writing an R package from scratch
  → Link (28/10/2020)

Programming style

A good programming style adheres to the following principles:

1. Exactitude
2. Readability
3. Comprehensibility
4. Reusability
5. Optimisation
6. Inspiration

Exactitude

Be as accurate as possible.

Typically, you should carry over results in their given precision and only start rounding for final tables etc.

Readability I

Elegant code might use uncommon functions and might be very curt. However, this type of efficiency should never hinder another user's understanding of the code. A good balance between these two poles should be found, resulting in code that is **neither too long nor too short**.
The **formatting** of the code itself can enhance its readability. Attention should be paid to the following formatting aspects:

1. **Indentation**
   Use consistent indentation of semantically connected code parts.
2. **Spacing**
   Use coherent spacing. This allows for a faster discrimination between different code parts, arguments, etc.
3. **Line length**
   As a rule of thumb, you should not use more than 80 characters (including spaces) per line. Having to scroll horizontally is annoying and time-consuming.

Readability II

```r
# Bad example
m<-matrix(1:100,nrow=10);s<-0;for(i in 1:10){
for(j in 1:10){if(m[i,j]%%2==0){s<-s+m[i,j]}}}; s
## [1] 2550
# Better example
m <- matrix(1:100, nrow = 10)
s <- 0
for (i in 1:10) {
  for(j in 1:10) {
    if (m[i, j] %% 2 == 0) {
      s <- s + m[i, j]
    }
  }
}
s
## [1] 2550
```

Readability III

```
# Even better
nat_numbers <- 1:100
position <- nat_numbers %% 2 == 0
sum(nat_numbers[position])
## [1] 2550
```


Comprehensibility

In this context, the following two aspects are worth mentioning:

1. **Meaningful naming**
   Use meaningful object and argument names that are descriptive of their contents.
2. **Comments**
   Use comments to structure your code and explain what is done.
   You would be surprised to learn just how much you can actually forget about your own code after several months have passed.

Reusability

You should **program your own functions** for recurring tasks that cannot be handled by already existing functions. This even applies within the context of a single script.

A major advantage of functions is that you can **fix potential bugs centrally** and do not have to update the same code chunk in multiple scripts, which is a rather nasty error source.

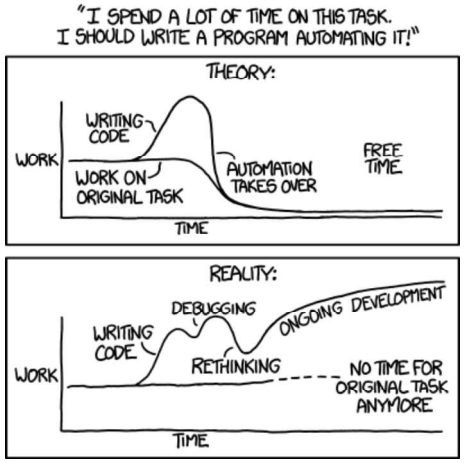
Try to **modularise your code**. Maybe a few of a number of functions can be re-used in other projects as well.

Optimisation

If you have to re-use code regularly and/or if the code takes a considerable amount of computation time, it is a good idea to **optimise this code** (making it run faster, adding *convenience features*, etc.).

However, mind the pitfalls of automation...

```
library(RXKCD); getXKCD(1319)
```



Source: xkcd.com (2020).

Inspiration

Get inspiration from frequently used existing functions in your field, be it across the globe or in your team. Chances are that the users of your functions will be familiar with certain required inputs, naming conventions, etc. This helps the acceptance of your functions and, consequently, their adoption.

Statistical Programming with R

Chapter 7 - Regression models and introduction to simulation

Florian Ertz, Jan Pablo Burgard, and Ralf Münnich

Economic and Social Statistics
Economics / Faculty IV
Trier University

Winter semester 2021/2022

Copyright statement

Data used

The book *Introductory Econometrics - A Modern Approach* by Jeffrey M. Wooldridge (as of 2020 in its seventh edition) is a good introductory text on regression modelling. In this chapter, we use data accompanying this text as compiled in the R package named `wooldridge` and some of its examples.

We start with the data set `wage1` that contains data from the US 1976 *Current Population Survey* and has been gathered by Henry Farber at MIT in 1988.

However, we first take a small detour.

Reading CSV files

We read comma-separated values (CSV) files using the function `read.csv`. Contrary to loading R data files, we should always combine this call with an assignment. Depending on the column and decimal separators used, you might also want to use `read.csv2` or set the arguments `sep` and `dec` as needed.

```
head(read.csv("Wage.csv"))
##   wage educ
## 1 3.10   11
## 2 3.24   12
## 3 3.00   11
## 4 6.00    8
## 5 5.30   12
## 6 8.75   16
some_data <- read.csv("Wage.csv")
```

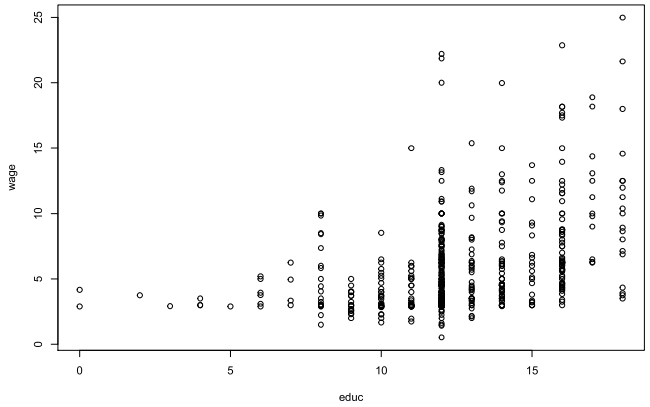

Reading text files

We read simple text files using the function `read.table`. If the text file contains column names, make sure to set its argument `header` to `TRUE`.

Wages and education

We now look at average hourly wages in USD (`wage`) and years of education (`educ`) of employees.

```
plot(wage ~ educ, data = some_data)
```



A simple linear regression model I

We now want to *formalise* the positive relation of these two variables we just saw using the following simple linear (regression) model:

$$wage = \beta_0 + \beta_1 \cdot educ + u,$$

where u captures other factors influencing the average hourly wage, i.e. the dependent variable, besides the number of years of education, i.e. the independent variable.

β_1 gives the (*ceteris paribus*) change in the average hourly wage induced by an additional year of education.

We now estimate the two regression parameters (betas) using the given sample data and the method of ordinary least squares (OLS).

A simple linear regression model II

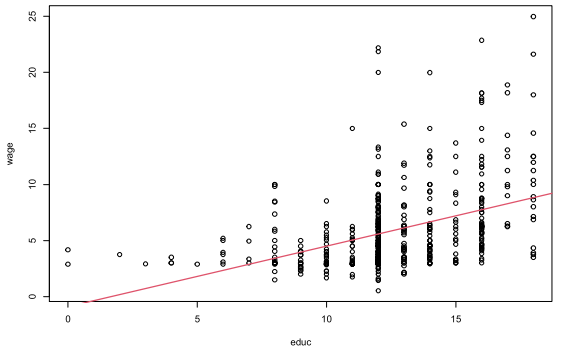
To estimate such a simple linear model using OLS, we use the function `lm`. The first argument is a formula. A call to our assigned linear model object (which is a list) yields the parameter estimates for the intercept ($\widehat{\beta}_0$) and the slope ($\widehat{\beta}_1$).

```
mod <- lm(wage ~ educ, data = some_data)
mod
##
## Call:
## lm(formula = wage ~ educ, data = some_data)
##
## Coefficients:
## (Intercept)         educ
##     -0.9049       0.5414
```


The (sample) regression line

We can now easily add the (sample) regression line that visualises the predictions for the dependent variable given certain values of the independent variable and our model to our original scatterplot.

```
plot(wage ~ educ, data = some_data)
abline(mod, col = 2, lwd = 2)
```



Predictions

To get predictions for specified values of the independent variable, like 10, 13, and 18 here, we use the function `predict`, which is another generic function by the way, and provide the linear model object as its first argument. But many ways lead to Rome...

```
predict(mod, newdata = data.frame(educ = c(10, 13, 18)))
##        1        2        3
## 4.508741 6.132819 8.839615
mod$coef[1] + mod$coef[2] * c(10, 13, 18)
## [1] 4.508741 6.132819 8.839615
```

Note that R *knows* that `coef` can only be short hand for the list element `coefficients`.

Residuals

The residuals ($\widehat{u}_i = y_i - \widehat{\beta}_0 - \widehat{\beta}_1 \cdot x_i$) are also saved in the linear model object.

```
head(mod$residuals, 5)
##          1          2          3          4          5
## -1.9501003 -2.3514594 -2.0501002  2.5739776 -0.2914593
```

ATTENTION:
The order of the residuals mirrors the order of the original data set.

t-test I

To get the variance estimates for the regression parameters and to perform significance tests on them, we use the function `summary` and provide the linear model object as its argument. The resulting object itself is once again a list that can be used to extract data, like the r squared.

```
sum_mod <- summary(mod)
head(names(sum_mod), 3)
## [1] "call"      "terms"     "residuals"
sum_mod$r.squared
## [1] 0.1647575
```


t-test II

```
sum_mod
##
## Call:
## lm(formula = wage ~ educ, data = some_data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.3396 -2.1501 -0.9674  1.1921 16.6085
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.90485    0.68497  -1.321    0.187
## educ         0.54136    0.05325  10.167   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '
##
```

t-test III

```
sum_mod$coefficients[, "Pr(>|t|)"] < 0.05
## (Intercept)         educ
##       FALSE         TRUE
```


A multiple linear regression model I

If we want to include more than one independent variable, we move from a single to a multiple linear (regression) model. We now *control* for these additional independent variables and prevent potential biases. Let us add the labour market experience of the sampled employees as another explanatory variable to our regression model.

```
#install.packages("wooldridge")
library(wooldridge)
mod_new <- lm(wage ~ educ + exper, data = wage1)
```


A multiple linear regression model II

Let us now predict the average hourly wage for an individual with 18 years of education and 10 years of labour market experience, without and with a confidence interval. The coefficient of determination has improved.

```
predict(mod_new, newdata = data.frame(educ = 18,
                                      exper = 10))
##        1
## 8.907312
predict(mod_new, newdata = data.frame(educ = 18,
                                      exper = 10),
        interval = "confidence"
        )
##        fit      lwr      upr
## 1 8.907312 8.291725 9.522899
summary(mod_new)$adj.r.squared
## [1] 0.2221991
```


Factors as explanatory variables I

To see how the function `lm` treats factors as explanatory variables, see the following example that uses the data set `Prestige` from the R package `car`.

```
#install.packages("car")
library(car)
## Loading required package: carData
head(Prestige$type)
## [1] prof prof prof prof prof prof
## Levels: bc prof wc
another_mod <- lm(prestige ~ education + income + type,
                  data = Prestige
                  )
```

Factors as explanatory variables II

```
summary(another_mod)$coefficients[, 1:2]
##                 Estimate   Std. Error
## (Intercept) -0.622929165 5.22752548777
## education    3.673166052 0.6405016204
## income       0.001013193 0.0002209185
## typeprof     6.038970651 3.8668550979
## typewc      -2.737230718 2.5139324035
```

If the variable `type` had not been a factor to start with, we would have had to construct dummy variables by hand first.

Binary variables as dependent variables

In economics and the social sciences, we do not only model metric variables (like hourly wages in the last section). Sometimes, we are interested in modelling binary variables as well, i.e. variables that only have two outcomes (0 and 1 or false and true). One example is the labour force participation of married women. Another example is whether a loan applicant gets granted the loan or not. In both cases, we are interested in the conditional probabilities of the event occuring, i.e. a woman being in the labour force or an applicant being granted a loan, respectively. The conditions are defined by a set of regressors.

Generalized linear regression models

In the linear model, the conditional expectation (μ_i) is equal to the linear predictor (η_i). When we move to conditional probabilities, like in the examples from the last slide, this does not hold anymore. Now the conditional expectation and the linear predictor are *linked* by a monotone transformation (the link function) g. We are dealing with a generalized linear (regression) model then.

There are also generalized linear models for categorical and count data, which we do not discuss here.

The logistic regression model

One example of a generalized linear model is the logistic regression or logit model. Using the conditional probability of success (p_i), this model is:

$$\text{Logit}(p_i) := \log\left(\frac{p_i}{1-p_i}\right) = \eta_i\,,$$

where the linear predictor η_i is defined as follows:

$$\eta_i = \beta_0 + \beta_1 \cdot x_{i1} + \beta_2 \cdot x_{i2} + \ldots + \beta_k \cdot x_{ik}\,.$$

To get from the linear predictor to the desired conditional probability of success, we have to use the inverse of the link function; in the logit case:

$$g^{-1}(\eta_i) = \frac{\exp(\eta_i)}{1+\exp(\eta_i)}\,.$$

Married women's labour force participation I

We now want to model the labour force participation of married women. The data set `mroz`, once again taken from the R package `wooldridge`, originates from the paper *The Sensitivity of an Empirical Model of Married Women's Hours of Work to Economic and Statistical Assumptions* published by T.A. Mroz in 1987 in *Econometrica* 55, pp. 765–799.

```
logit_mod <- glm(inlf ~ nwifeinc + educ + exper +
                        exper^2 + age + kidslt6 +
                        kidsge6,
                 data = mroz,
                 family = binomial(link = "logit")
                 )
```

Note that we use the function `glm` instead of the function `lm` and the argument `family`.

Married women's labour force participation II

```
summary(logit_mod)$coef
##                Estimate Std. Error    z value     Pr(>|z|)
## (Intercept)  0.83790883 0.84093267  0.9964042 3.190538e-01
## nwifeinc    -0.02021649 0.00826364 -2.4464387 1.442753e-02
## educ         0.22697664 0.04329514  5.2425427 1.583787e-07
## exper        0.11974577 0.01362629  8.7878497 1.524497e-18
## age         -0.09108842 0.01432061 -6.3606541 2.008964e-10
## kidslt6     -1.43939286 0.20149766 -7.1434720 9.100248e-13
## kidsge6      0.05817351 0.07337970  0.7927739 4.279096e-01
```

ATTENTION:
Note that the point estimates cannot be interpreted as marginal effects, like in the linear model, anymore.

Linear predictors and predicted probabilities

We can get the linear predictors as well as the predicted probabilities from the model.

```
new_pred <- data.frame(nwifeinc = mean(mroz$nwifeinc) * 2,
                       educ = 18, exper = 5,
                       age = 35, kidslt6 = 2,
                       kidsge6 = 0
                       )
predict(logit_mod, newdata = new_pred)
##         1
## -1.358537
predict(logit_mod, newdata = new_pred, type = "response")
##         1
## 0.2044782
```


Percentage of correct *predictions*

To assess the quality of our model, we can compute the share of correct *predictions*.

```
(pred_tab <- prop.table(
             table(Obs = mroz$inlf,
                   Preds = round(fitted(logit_mod))
                   )
             ))
##    Preds
## Obs         0         1
##   0 0.2815405 0.1500664
##   1 0.1075697 0.4608234
sum(diag(pred_tab)) / sum(pred_tab)
## [1] 0.7423639
```

Core idea of a simulation study

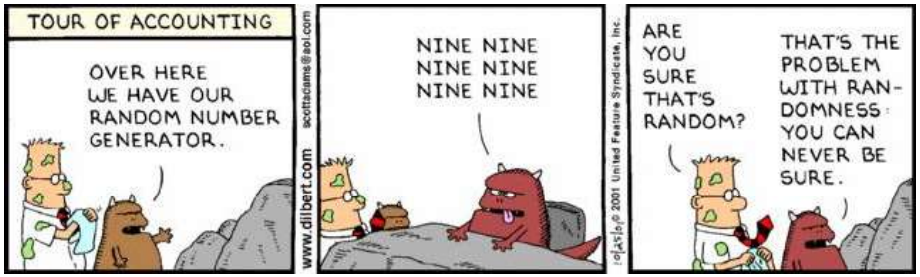
In this section, we illustrate the core idea of a Monte Carlo simulation study with the help of a simple linear model. We would like to demonstrate that the OLS point estimator for the slope is unbiased if the classical linear model assumptions hold. Accordingly, we need to approximate the expected value of the respective point estimator. To that end, we first construct a simulation population.

Pseudo-random number generation

Let our simulation population contain $N = 10,000$ observations of pairs of metric variables X and Y that are functionally related. We use R's built-in pseudo-random number generation. To generate random variates, we use functions whose names combine the prefix `r` and an abbreviation for the probability distribution from which we would like to draw. For example, to generate random variates from a normal distribution, we use the function `rnorm`. To ensure reproducibility, we first set a starting value for the pseudo-random number generator with the function `set.seed`.

```
N <- 10000
set.seed(123)
x <- rnorm(n = N, mean = 10, sd = 2)
betas <- c(1, 3)
set.seed(124)
errors <- rnorm(N, 0, 8)
y <- cbind(1, x) %*% betas + errors
```


Random numbers



Source: Dilbert by Scott Adams (2001); Link (06/01/2021).

Probability distribution functions

Most relevant probability distributions are implemented in R and can be used with the help of the following four functions, where every prefix is followed by an abbreviation of the distribution name:

1. **p** (for **probability**):
   Cumulative distribution function (CDF)
2. **q** (for **quantile**):
   Inverse CDF
3. **d** (for **density**):
   Density function
4. **r** (for **random**), as above:
   (Pseudo-)random variates from distribution

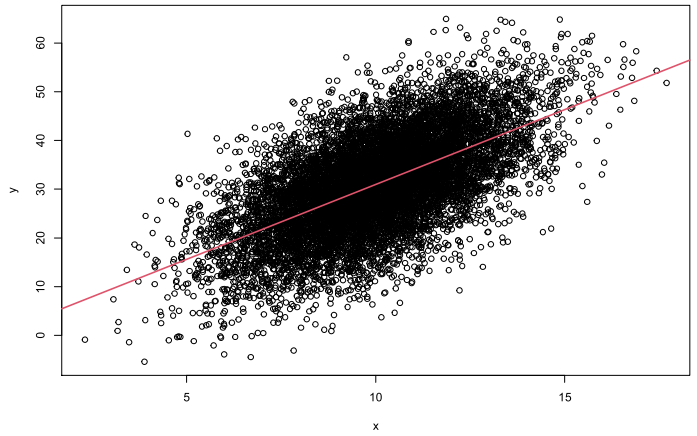
Examples of built-in probability distributions

Below, you find a selection of probability distributions that come with R's base installation.

Distribution	Functions			
Beta	pbeta	qbeta	dbeta	rbeta
Binomial	pbinom	qbinom	dbinom	rbinom
Chi-square	pchisq	qchisq	dchisq	rchisq
Exponential	pexp	qexp	dexp	rexp
F	pf	qf	df	rf
Gamma	pgamma	qgamma	dgamma	rgamma
Geometric	pgeom	qgeom	dgeom	rgeom
Hypergeometric	phyper	qhyper	dhyper	rhyper
Logistic	plogis	qlogis	dlogis	rlogis
Log-normal	plnorm	qlnorm	dlnorm	rlnorm
Normal	pnorm	qnorm	dnorm	rnorm
Poisson	ppois	qpois	dpois	rpois
Student t	pt	qt	dt	rt
Uniform	punif	qunif	dunif	runif

Simulation population and population regression line

```
plot(x, y)
pop_mod <- lm(y ~ x)
abline(pop_mod, col = 2, lwd = 2)
```



Sampling

We now draw $R = 2,000$ samples of size $n = 200$ from the population. To do simple random sampling (SRS) with or without (the default) replacement, we use the function `sample`.

```
R <- 2000
n <- 200
set.seed(125)
units <- lapply(rep(n, R), function(x)
                sample(1:N, x, replace = TRUE)
                )
head(units[[1]])
## [1] 4894 7658 3263 6984 2169 5385
head(units[[2]])
## [1] 5473 7371  252 4710 6250 9450
```


Modelling in samples

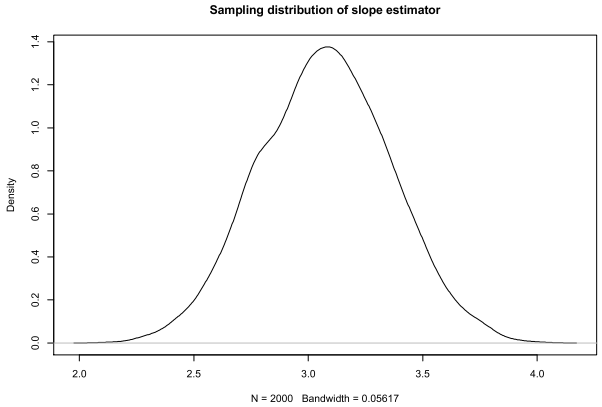
We now estimate a linear model in each of the $R = 2,000$ samples and investigate the resulting sampling distribution of the slope estimator.

```
dat_df <- data.frame(x = x, y = y)
sample_slopes <- sapply(units, function(z)
                        lm(y ~ x, data = dat_df[z, ])$coef[2]
                        )
summary(sample_slopes)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   2.144   2.876   3.076   3.073   3.267   4.003
pop_mod$coef
## (Intercept)           x
##   0.2458721   3.0732572
```

Sampling distribution

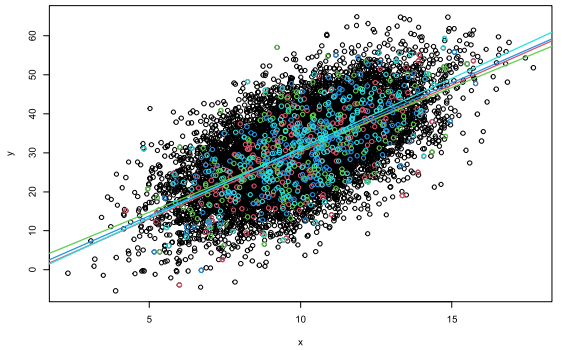
We now graph the sampling distribution.

```
plot(density(sample_slopes),
     main = "Sampling distribution of slope estimator")
```



Samples and sample regression lines

```
set.seed(126); samples2 <- units[sample(1:R, 4)]; plot(x, y)
lapply(1:4, function(x) {
 points(dat_df[samples2[[x]], ], col = x + 1)
 abline(lm(y ~ x, data = dat_df[samples2[[x]], ]),
        col = x + 1, lwd = 2)})
```



Approximating π I

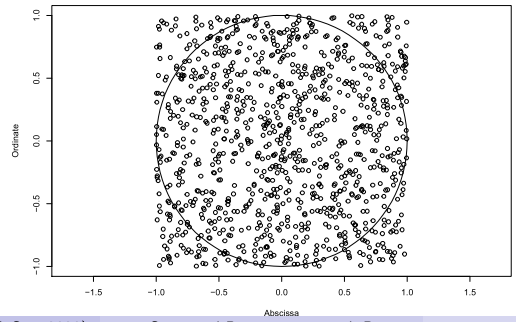
To approximate the area of a unit circle, i.e. π, we can use the fact that it is closely linked to the probability of being within a unit circle inside a 2x2 square.

Accordingly, in a small Monte Carlo experiment we can compute the percentage of points within the unit circle and multiply it by 4.

Approximating π II

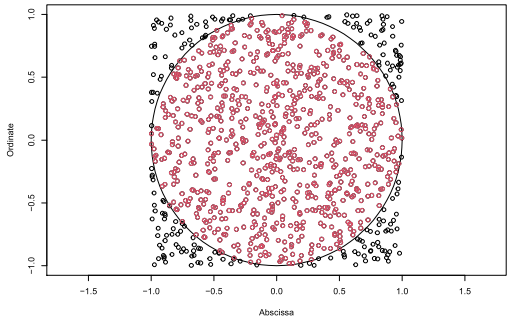
We use random variates of a uniform distribution on $[-1, 1]$.

```
n <- 1000; set.seed(127)
Abscissa <- runif(n, -1, 1); Ordinate <- runif(n, -1, 1)
#install.packages("plotrix")
library(plotrix); plot(Abscissa, Ordinate, asp = 1)
draw.circle(0, 0, 1)
```



Approximating π III

```
inside <- (Abscissa^2 + Ordinate^2) <= 1
plot(Abscissa, Ordinate, asp = 1)
draw.circle(0, 0, 1)
points(Abscissa[inside], Ordinate[inside], col = 2, lwd=1.5)
```




```
mean(inside) * 4
## [1] 3.144
```