Runifr - A gentle Introduction to R

Wegmann Lab - UniFR

2021-10-08

Contents

Overview

Course organisation

This course introduces important concepts of the R programming language and its usage. It is designed as an introductory course for people with limited programming experience, but with the goal to make you fit for follow-up courses based on R such as courses in advanced statistics, data science or bioinformatics. Or to get you started exploring the endless possibilities of R on your own.

The material is used at the University of Fribourg in multiple ways:

Winter School in Data Analytics and Machine Learning

The module **Introduction to R** covers Chapters 1-12

MSc in Bioinformatics & Computational Biology / MSc in Data Analytics & Economics

Students in these MSc programs take the full course, organized in two modules:

- **SBL.30001 Introduction to R**: This course covers the material presented in Chapters 1-12.
- **SBC.07109 Programming in R**: This course covers the material presented in Chapters 13-15.

MSc in MolcecularLife & Health Sciences / MSc in Environmental Biology

Students of these MSc programs need to take the first module

- **SBL.30001 Introduction to R**: This course covers the material presented in Chapters 1-12.

Chapter 1

Getting Started

What is R?

R is an environment designed for data manipulation, statistical computing and graphical display. It is by far the most popular programming language for statistics and particularly widely used in biology any many other fields.

In contrast to other statistical systems, a statistical analysis in R is usually done as a series of steps with intermediate results stored in objects. While maybe less convenient that on-click software (e.g. SPSS), it allows any analysis to be **customized** and **scripted**, and thus in turn to be **automatized** and **reproduced**.

Using R

R is an **interpreted language**, which means that instructions are executed directly at run time through a command-line interpreter. Or in plain English: R is mostly used via a console in which you can type commands that are then interpreted and executed.

A particularly convenient way to use R is through RStudio, a dedicated IDE (integrated development environment) made specifically for R. In this course, we will mostly use RStudio, but also discuss how to run R scripts from the command line towards the end.

1.1 Installing RStudio

To get ready to learn R, hop over to rstudio.com and install R on your system. RStudio is available for Linux, Windows and Mac and should be trouble free to install. Let's go!

1.1.1 Getting familiar with RStudio

After installing RStudio, launch it and you will be greeted by something like this:

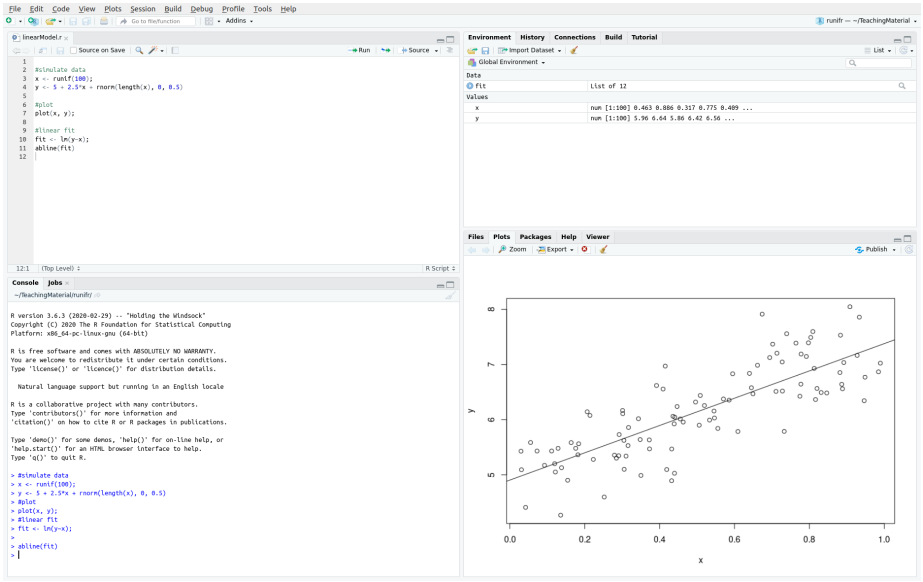


Figure 1.1: Rstudio interface

Note that RStudio has four panes:

- The running R console, which is on the lower left in the image above.
- A pane to write source code (i.e. R scripts), which is on the top left.
- Two extra panes with tabs for plots, help files, the environment, history, ... on the right.

The placement of these panes is customizable, they can be resized with your mouse, and a pane might be minimized if not used. We will learn how to make use of them later. For now, just make sure you identify the console pane which we will use first. Found it? Then you are ready to use R!

Chapter 2

Basic Syntax

To get started, let's use R as a simple calculator. To try it out, navigate to the console pane of RStudio and try adding two numbers by typing `5+8` into your console and press enter. You should see something like this:

```
> 5+8
[1] 13
```

The `>` indicates that the console is ready to accept input. The result is printed directly into the console with a preceding `[1]`. This indicates an index as by default R treats every number is a vector of length 1. We will talk about vectors later - for now you can safely ignore the `[1]`.

Execution is triggered by both a newline (e.g. pressing enter) or a semicolon (`;`):

```
> 1+1; 2-4
[1] 2
[1] -2
```

R offers all standard arithmetic operators, including the basic operators `+`, `-`, `*` and `/` as well as `^` for exponents and `%%`for modulus (the remainder from division). Here are some examples

```
> 5*3
[1] 15
> 2^10
[1] 1024
> 7%%3
[1] 1
```

R respects the standard order of operations such that multiplications (and divisions) are given a higher precedence than additions (and subtractions) and exponents precede both. To choose a specific order, parenthesis may be used.

```
> 1+2*3
[1] 7
> (1+2)*3
[1] 9
> 1+2^3
[1] 9
> (1+2)^3
[1] 27
```

2.0.1 Exercises: Arithmetic Operators

See Section 18.0.1 for solutions.

Calculate the following expressions:

1) $1 + 2 + 3$

2) $\frac{1}{1+2}$

3) 2^{1+2}

2.1 Functions

A central part of any programming language are functions. Functions take some input to generate output. In mathematics, $f(x)$ denotes a function f that takes an input argument x, does something with it, and returns the result to us. Examples for well known functions are:

- the logarithmic function: $\log(x)$
- the exponential function: e^x
- the square root: $\sqrt{x}$
- the sinus function: $\sin(x)$

R has a large number of built-in functions that we can readily use. Just pick a number, say 5, and type into the console:

```
> log(5)
[1] 1.609438
> exp(5)
[1] 148.4132
> sqrt(5)
[1] 2.236068
> sin(5)
[1] -0.9589243
```

As you can see, functions in R consists of a name, e.g. `log`, and parenthesis (i.e. round brackets), `()`. The arguments for the function are specified inside the parenthesis, e.g. `log(5)`, where `5` is the argument to the function `log`.

Some functions can take multiple arguments. An example is the logarithmic function `log()`. If we only specify one argument, e.g. 5, it will compute the natural logarithm:

```
> log(5)
[1] 1.609438
```

The natural logarithm uses base $e = 2.718282....$ The base to use can be provided to the function `log` as a second argument. If we provide the base `exp(1)` as a second argument, we get exactly the same result:

```
> log(5, exp(1))
[1] 1.609438
```

This second argument is said to have a **default value** that is used if we don't specify anything else. However, we can also pick another logarithmic base, for example 10, and override the default argument:

```
> log(5, 10)
[1] 0.69897
```

In R, every argument has a name. We can either pass the arguments with or without name to the function:

```
> log(5, 10)
[1] 0.69897
> log(5, base = 10)
[1] 0.69897
> log(x = 5, base = 10)
[1] 0.69897
```

When providing a name, we can even shuffle the arguments around:

```
> log(base = 10, x = 5)
[1] 0.69897
```

which still gives the same result, because R knows which value belongs to `base` and which one belongs to `x`. If we don't specify the argument names, R would interprets `x = 10` and `base = 5`,

```
> log(10, 5)
[1] 1.430677
```

which is obviously not the same.

2.1.1 Help pages

But how do you even know which function to use, what a function exactly does or which arguments it takes? Fortunately, there are many ways to get help!

For starters, there is an internal help-page for every function. For getting help on the `log()`, just type into the console:

```
> help(log)
```

or the shortcut

```
> ?log
```

Sometimes, you might not remember the exact name of the function. You can then browse through the help files using

```
> help.search("logarithm")
```

or the shortcut

```
> ??logarithm
```

These built-in help functions are useful if you already know quite specifically what you're looking for. However, you often have only a vague idea on how to solve your problem. In this case, internet search engines are your best friend. Knowing how to "google" is a very important skill in programming, and can be trained as every other skill. Try to describe your problem in a short, abstract way, and add the name of your programming language (in our case R) to the search. Or, if you have an error you don't understand, paste the entire error message into the search field. On websites like stackoverflow.com, you practically always find someone who solved the problem you encountered.

2.1.2 Exercises: Functions

See Section 18.0.2 for solutions.

1) Search for the function `round()` by using the R help page. What is the purpose of this function? Which argument(s) does it take? Which argument(s) have a default value?

2) Round ´1.23456789´ to zero, three and seven decimal places. Try calling the function with and without providing argument names.

3) Try to guess the result of `round(5.678, digits=1)`, `round(x=5.678, digits=1)`, `round(digits=1, x=5.678)` and `round(1, 5.678)`!

4) Find the function to calculate the binomial coefficient $\binom{5}{3}$.

2.2 Variables

Programming would be impossible without variables that allow us to store data for later use. Their value can be changed according to your need.

The recommended operator to assign data to a variable is `<-`. For example, you can assign the value `7` to `x` by typing

```
> x <- 7
```

Note that the object `x` is created at the same time as the assignment, there is no need to define it first. But the same operator can also be used to overwrite the content of a variable. To illustrate that, we will use the function `print()` which prints the content of a variable.

```
> x <- 7
> print(x)
[1] 7
> x <- 5
> print(x)
[1] 5
```

As a helpful shortcut to `print()`, you may just type the name of a variable, which will call the function `print()` with default arguments.

```
> x <- 7
> x
[1] 7
```

A valid variable name in R consists of letters, numbers and the dot or underline characters. You can not name a variable that starts with a number. Also, variable names in R are case sensitive, so `x <- 7` and `X <- 7` are treated as two different things.

You can use variables to perform calculations the same way you can type numbers into the console:

```
> a <- 4
> b <- 7
> a+b
[1] 11
> a/b
[1] 0.5714286
```

Note that variables can also be assigned using the `=` operator as in

```
> x = 7
> x
[1] 7
```

However, it is best practice to only use `<-` as the `=` can have multiple meanings

in R.

2.2.1 Variable Names

Once you start working in R, you will notice that the amount of variables you are working with is adding up quite quickly. It is therefore important to give your variables sensible names that you will understand even if you have not looked at your code for a while. And while we used such names here frequently to illustrate concepts, `x`, `a` or `b` are not very telling names.

In fact, telling names are usually rather long and often consist of multiple words. Since R does not allow spaces within variable names, people generally use the underscore (`random_number <- 856.5`) or dot (`random.number <- 856.5`) to separate words, or use so-called camel case (`randomNumber <- 856.5`) for their variables. Of those, both Google and the tidyverse community recommend using underscores (`_`) for variable names as dots (`.`) have a special meaning in S3 classes (see MUCH later) and camel cases is preferred for functions. So why not stick to that?

2.2.2 Exercises: Variables

See Section 18.0.3 for solutions.

1) Assign the values 6.7 and -56.3 to variables `value_1` and `value_2`, respectively.
2) Use `help.search()` to find out how to compute the square root of variables and compute the square root of `value_1` and `value_2`.
3) Use R to calculate $\frac{2a}{b} + ab$ using `value_1` for a and `value_2` for b and assign the results to variable `x`.

2.3 Console pane tricks

R is unforgiving: even the slightest spelling mistake will cause an error. It is therefore not uncommon to experiment with a command before it does exactly what you want. RStudios' console pane offers several nice features to make that a little less painful.

2.3.1 Browsing past commands

A particularly helpful feature: you can use the arrow keys to easily retrieve a previously typed command. The arrow-up button lets you browse your command-history backward. Press it once, and you'll see your last command. Hit it a second time, and the before-last command pops up. Neat, right?

In case you passed the command you were looking for, you can use the arrow-down key to browse it forward. Try it out!

In addition, you can trigger a drop-down menu that lists all your past commands with `ctrl` + the arrow up key.

2.3.2 Auto-completion

RStudio provides auto-compleition suggestions while typing. Just type part of a command and pause a little anda drop-down menu with suggestions shows up. From it, you can choose the function or variable name you are looking for and hit return. For instance, assume you declared the variable

```
> terribly_long_variable_name_I_can_never_spell_correctly <- 10
```

If you now type `terr` a wait for a second, that horribly long variable name shows up in the dropdown for you to choose. And rather than using your arrow keys to choose an option, you can also continue typing until the first choice is what you were looking for and then hit return.

Good to know: if you are impatient or the auto-compleition dialog does not show up, just hit `ctrl` + `space` to trigger it.

Chapter 3

R-Scripts

As your analysis conducted in R gets more complicated (at least in the number of expressions to evaluate), you may want to save your work as an R script, rather than typing one command after the other into the console. An R Script is really just a plain text file containing R code that can be executed. A major benefit of R scripts is that you preserve the analysis you did and can easily re-run it again - that is not only going to save you time, but it will also make your work reproducible.

RStudio is fantastic for working with R scripts. Just click on File -> New File -> R Script in the menu and a new file will be opened in the source pane. There, you can type your R code and save it for later use, but also send it to the console for execution.

For instance, copy the following code into an empty R script:

```
x <- 10
y <- 20
z <- x + y
```

You can now execute this R-script in one of two ways in RStudio:

- You can send line by line to the console. for this, place the cursor anywhere on the chosen line an click the "Run" button (or press `Ctrl` + `Enter` as a shortcut). The cursor will automatically jump to the next line, so you can just keep pressing the "Run" button (or pressing `Ctrl` + `Enter`) until you reach the end of the script.
- Alternatively, you can ask for the whole document to be executed. For this, simply press the "Source" button (or press `Ctrl`+ `Shift`+ `Enter` as a shortcut).

Cool, no?

Comments

As your R code will grow, it will become very important to comment it properly. Indeed, a variable `day_1` will maybe mean a a lot to you while writing your code, but will you remember its meaning when going back to it after one year? Similarly, you maybe side with a particular statistical analysis, but will you remember the reasoning a year later?

Most peopel don't - which is why most people should add comments to their code. In R, comments always start with a `#` symbol. This tells R that everything after the `#` is not R code and is to be ignored.

```
> x <- 7
> #x <- 5
> x
[1] 7
```

You can add comments at the end of a line too

```
> num_replicates <- 10 # I chose 10 to balance statistical power with sampling effort
```

Since R scripts can get very long, you can also use comments to structure them or leave other worthy notes. Here is an example:

```
#------------------------------------------------
# My first R script
#
# This script illustrates what I learned so far
#------------------------------------------------

# Assigning variables
num_replicates <- 4
b <- 8 # TODO: This is a bad variable name - think of a better one

# Calling functions
z <- log(a) - log(b);
print(exp(z)) # we expect this to be 0.5
[1] 0.5
```

3.0.1 Exercises: R-Scripts

See Section 18.0.4 for solutions.

1) Create an R-script that assigns the values $\pi = 3.1415926\ldots$, 10 and $e = 2.718282\ldots$ to variables `my_data`, `base_1` and `base_2` and then prints the log of `my_data` with both bases. Use comments to explain your script and save it as `myFirstRScript.r` on your computer.

2) Execute your script `myFirstRScript.r` both line-by-line as well as in one go (i.e. source it).

Chapter 4

Vectors

Vectors are the most basic R data objects. Even when you write a single number in R, it becomes a vector of length one:

```
> 5
[1] 5
```

The preceding `[1]` that is printed to the console indicates the index of the vector. In this case, 5 is the first element in the vector and therefore has index 1.

A vector can have more than one element. Formally, it is a sequence of elements of the same type, for example numbers. Vectors can be created using the `c()` function:

```
> x <- c(2, 4, 6, 8, 10)
```

This function `c()` simply concatenates all arguments into a single vector. It can also coerce vectors and numbers or vectors and vectors:

```
> x <- c(2, 4, 6, 8, 10)
> c(-100, x, 100)
[1] -100    2    4    6    8   10  100
> c(c(2, 4, 6), c(8, 10))
[1]  2  4  6  8 10
```

The number of elements in the vector is obtained by the function `length()`.

```
> x <- c(2, 4, 6, 8, 10)
> length(x)
[1] 5
```

4.0.1 Sequences

Sequences of numbers are used frequently, and R offers functions to create such sequences easily. For creating a vector of consecutive numbers, we can use the `:` operator:

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> x <- 10; y <- 5
> x:y
[1] 10  9  8  7  6  5
> (x-1):(y+2)
[1] 9 8 7
```

The function `seq()` offers a more general way to create sequences of numbers. The first two arguments specify the beginning and end of the sequence.

```
> seq(1, 10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

The third argument can either be `by`, denoting the step size, or `length.out`, denoting the total length of the sequence:

```
> seq(-5, 5, by = 0.5)
 [1] -5.0 -4.5 -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0
[16]  2.5  3.0  3.5  4.0  4.5  5.0
> seq(-5, 5, length.out = 21)
 [1] -5.0 -4.5 -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0
[16]  2.5  3.0  3.5  4.0  4.5  5.0
```

The function `rep()` can be used for replicating an object in various ways.

```
> rep(5)
[1] 5
> rep(5, times=3)
[1] 5 5 5
> rep(5, length.out=3)
[1] 5 5 5
```

`rep()` can also replicate vectors with more than one element. When specifying `times`, it will replicate the entire vector this many times. `times` can also be a vector itself (of same length as `x`), in which case it will replicate the each element of `x` according to the corresponding element in `times`:

```
> x <- c(2, 4, 6, 8, 10)
> rep(x, times = 3)
 [1]  2  4  6  8 10  2  4  6  8 10  2  4  6  8 10
> rep(x, times = 1:5)
 [1]  2  4  4  6  6  6  8  8  8  8 10 10 10 10 10
```

When specifying `length.out`, it will replicate the elements of the vector as long as this length is achieved:

```
> x <- c(2, 4, 6, 8, 10)
> rep(x, length.out = 6)
[1]  2  4  6  8 10  2
```

When specifying `each`, it will repeat each elements this many times consecutively:

```
> x <- c(2, 4, 6, 8, 10)
> rep(x, each = 2)
 [1]  2  2  4  4  6  6  8  8 10 10
```

These arguments can be combined in many ways:

```
> x <- c(2, 4, 6, 8, 10)
> rep(x, length.out = 6, each = 2)
[1] 2 2 4 4 6 6
> rep(x, times = 3, each = 2)
 [1]  2  2  4  4  6  6  8  8 10 10  2  2  4  4  6  6  8  8 10 10  2  2  4  4  6
[26]  6  8  8 10 10
```

4.0.2 Accessing Elements of Vectors

Elements of a vector can be accessed by indexing. Indexing in R starts at 1, i.e. the first element in the vector has index 1. Most other programming languages, including python, start indexing at 0, so be aware of this! We can access the values inside a vector by declaring the index inside the square brackets `[]`:

```
> x <- c(2, 4, 6, 8, 10)
> x[1]
[1] 2
> x[2]
[1] 4
> x[5]
[1] 10
```

We can access more than one element at once (this is called slicing) by providing a sequence of indices:

```
> x <- c(2, 4, 6, 8, 10)
> x[2:4]
[1] 4 6 8
```

In addition, this sequence of indices doesn't have to be consecutive:

```
> x <- c(2, 4, 6, 8, 10)
> x[c(1, 3, 5)]
[1]  2  6 10
```

We can also provide negative indices. In this case, it just removes the element with this (absolute) index from the vector:

```
> x <- c(2, 4, 6, 8, 10)
> x[-3]
[1]  2  4  8 10
> x[-(2:4)]
[1]  2 10
```

Apart from just reading values, indexing can also be used to set values inside a vector:

```
> x <- c(2, 4, 6, 8, 10)
> x[1] <- 100
> x[3:5] <- 1:3
> x
[1] 100   4   1   2   3
```

4.0.3 Integer Vectors

Each vector in R is of a specific type. All examples so far were of type `integer` as they only contained integers. You can check the type using the function `class()`.

```
> class(1)
[1] "numeric"
> class(-3:3)
[1] "integer"
```

The function `integer()` creates an integer vector of the specified length. The elements of the vector are all equal to 0.

```
> integer(10)
 [1] 0 0 0 0 0 0 0 0 0 0
```

In R, there are six atomic data types that are used to build atomic vectors:

- character: e.g. `'a'`, `"b"`, `"hello"`
- numeric: e.g. `-1`, `3.46`
- logical: `FALSE`, `TRUE`
- integer: e.g. `3L` (the `L` for "long" tells R that this is an integer)
- complex: `1+6i` (complex numbers with real and imaginary parts)
- raw: contains raw bytes; e.g. "hello" corresponds to 68 65 6c 6c 6f

In the following we will introduce the most frequently used types in more detail: `numeric`, `logical`, `character` and `factors`.

4.0.4 Exercises: Introduction to vectors

See Section 18.0.5 for solutions.

1) Create a vector `x` with values -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5 using `c()`.
2) Given `x` from above, append -6 in front of the vector and 6 at the end.
3) Create `vec1` with values -5, -4, ..., 4, 5 and `vec2` with values 6, 7, ..., 10. Now concatenate them. What is the length of the resulting vector? What are its elements?
4) Create the following sequences using the `:` operator: -5, -4, ..., 4, 5 and 10, 9, ..., 6.
5) Create the following vectors with the function `rep()`:
   - `vec1` with values 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
   - `vec2` with values 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5
   - `vec3` with values 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5
   - `vec4` with values 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5
6) Create the following vectors with the function `seq()`:
   - `vec1` with values 0.0, 0.5, 1.0, ..., 5.0
   - `vec2` with values 1.0, 2.4, 3.8, 5.2, 6.6, 8.0
   - `vec3` with values 10.0, 9.2, 8.4, ..., -0.4
7) Create a vector `x` with values -5, -4, -3, ..., 4, 5. Then, use indexing to access the element with value 2.
8) Access all elements of `x` with even values.
9) Set the last 3 elements of `x` to the values 30, 40, and 50, respectively.
10) Create an empty vector of size 5 using `numeric()`. Then, use indices to set the first element to 1, the second to 2 etc.
11) Given a vector `x` with values -5, -4, -3, ..., 3, 4, 5, exclude all elements with even values.

4.1 Numeric vectors

A vetcor containing decimal numbers (e.g. 3.1415926) is of type `numeric`.

```
> class(3.1415926)
[1] "numeric"
```

Note that R derives the data type of the variable implicitly from the object assigned to the variable. In the case of `x <- 3.46`, for instance, we don't need R to tell that `3.46` is numeric - it will automatically infer this itself.

Just as integer vectors, numeric vectors are also created using the `c()` function, or by using the function `numeric()` that creates a numeric vector of specified length with all elements set to zero.

```
> c(1.5, 2.0)
[1] 1.5 2.0
> numeric(10)
 [1] 0 0 0 0 0 0 0 0 0 0
```

Note that since vectors can only hold elements of a single type, the `c()` function coerces all elements to a common type.

```
> x <- c(1,5)
> class(x)
[1] "numeric"
> x
[1] 1 5
> x <- c(x, 2.5)
> class(x)
[1] "numeric"
> x
[1] 1.0 5.0 2.5
```

You can coerce vectors to a specific type using the `as.integer()` or `as.numeric()` function.

```
> as.numeric(1:3)
[1] 1 2 3
> as.integer(c(-1.8, 2.8))
[1] -1  2
```

As you can see, an integer vector can only hold integer numbers and a coercing of decimal numbers results in a loss of the information past the decimal point (no, the numbers are not rounded).

4.1.1 Arithemtic Operations with Vectors

Numeric vectors can be used in arithmetic expressions, in which case the operations are performed element by element:

```
> x <- 1:4
> y <- c(1.1, 1.2, 1.3, 1.4)
```

```
> x+y
[1] 2.1 3.2 4.3 5.4
> x^y
[1] 1.000000 2.297397 4.171168 6.964405
```

If two vectors are of different length, the shorter of the two is recycled. This means that the values from the shorter vector are simply re-used until the length of the longer one is reached. If R can not fully recycle the shorter vector, it will display a warning message but still do the operation.

```
> x <- 1:4
> y <- 0:1
> x*y
[1] 0 2 0 4
> y <- 0:2
> x*y
Warning in x * y: longer object length is not a multiple of shorter object
length
[1] 0 2 6 0
```

Recycling appears in many contexts, and it's good to know about it. For example, we can make use of recycling to set all even numbers to 0:

```
> x <- 1:20
> x * c(1, 0)
 [1]  1  0  3  0  5  0  7  0  9  0 11  0 13  0 15  0 17  0 19  0
```

... and much more!

Apart from elementary arithmetic operators (`+`, `-`, `*`, `/`, `%%`, `^`) discussed above, all of the common arithmetic functions are available, for example `log()`, `exp()`, `sin()`, `cos()`, `tan()`, `sqrt()` etc. These functions operate element-by-element and return a vector of same length as the input vector:

```
> x <- 1:10
> log(x)
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
> sqrt(x)
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
 [9] 3.000000 3.162278
```

Other functions take the whole vector as an argument, rather than applying the function to each element. These functions then return a single value. Examples are `mean()`, `sd()`, `var()`, `median()`, `min()`, `max()`, ...

```
> x <- 1:10
> mean(x)
[1] 5.5
```

```
> sd(x)
[1] 3.02765
> var(x)
[1] 9.166667
> median(x)
[1] 5.5
> min(x)
[1] 1
> max(x)
[1] 10
```

Note that R automatically coerces types where necessary. This is illustrated in the following example: when adding two integer vectors, the result is also an integer vector. But when adding an integer and a numeric vector, the result is a numeric vector. Similarly, when dividing two integer vectors, the result is of type numeric.

```
> class(1:5 + 1:5)
[1] "integer"
> class(1:5 + 0.5)
[1] "numeric"
> class(1:10/2)
[1] "numeric"
```

4.1.2 Exercises: Numeric vectors

See Section 18.0.6 for solutions.

1) Calculate the differences, sums and product between the elements of vectors `a`, containing the values from -5 to 5, and `b`, containing values from 10 down to 0, respectively.

2) Calculate the sum of all elements in `a` and `b`.

3) Identify the largest and smallest values of `a` and compute the overall mean.

4) Standardize `a` such that it has mean 0 and sd 1. Test your googling-skills if you're not sure how to do this.

5) Use recycling to multiply every 3rd element of `a` with 5. Why do you get a warning message?

4.2 Logical vectors

So far we looked at `integer`and `numeric` vectors. In this section, we will take a closer look at logical vectors.

Logical vectors contain `FALSE` and `TRUE` values. We call `FALSE` and `TRUE` booleans (or bools), which is a data type that has only 2 values, i.e. false and true.

We can assign a bool to a variable just like before with numbers: `a <- TRUE; b <- FALSE`. However, logical vectors are usually generated through conditions. For example

```
> x <- 1 < 2
> x
[1] TRUE
> y <- 2 < 1
> y
[1] FALSE
```

sets `x` and `y` as a vector of length 1 with values `FALSE` corresponding where the condition is not met and `TRUE` where it is. Logical operators, `<`, `<=`, `>`, `>=`, `==` and `!=`, are used for such conditions:

```
> 1 < 2
[1] TRUE
> 1 <= 2
[1] TRUE
> 1 > 2
[1] FALSE
> 1 >= 2
[1] FALSE
> 1 == 2
[1] FALSE
> 1 != 2
[1] TRUE
```

Here, ==corresponds to "equal" and `!=` corresponds to "not equal". If you have not programmed before, you maybe find the use of `==` rather that `=` strange to test for equality. But in most programming languages the `=` is reserved for assignments - including in R even if its use is discouraged.

```
> a <- 5
> a == 7
[1] FALSE
> a = 7
> a
[1] 7
```

In addition, two logical vectors can be compared with the logical operators `&` (intersection, "and"), `|` (union, "or") and `!` (negation, "not"):

```
> x <- TRUE
> y <- FALSE
> x & y
```

```
[1] FALSE
> x | y
[1] TRUE
> !y
[1] TRUE
```

`&` returns true if both conditions are true; `|` returns true if at least one condition is true; and `!` simply returns the opposite of the value.

Of course, a logical vector can have more than just one element. In this case, when comparing two logical vectors with `&` and `|`, comparison goes element-by-element.

```
> x <- 1:10 > 4
> x
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> y <- 1:10 <= 6
> y
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
> x & y
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
> x | y
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> !y
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

You will sometimes encounter the logical operators `&&` and `||`. The difference to `&` and `|` is that `&&` and `||` only compare a single element. If you compare two vectors of length > 1, this means that only the first element of the vectors is compared:

```
> x <- 1:10 > 4
> y <- 1:10 <= 6
> x && y
[1] FALSE
> x || y
[1] TRUE
```

We can use logical vectors for accessing elements of other vectors. Values corresponding to `TRUE` in the logical vector are selected and those corresponding to `FALSE` are omitted.

```
> x <- 1:5
> x[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
[1] 1 3 5
> x[rep(c(TRUE, FALSE), length.out = 5)]
[1] 1 3 5
```

This is particularly helpful to select only elements matching a certain condition.

```
> x <- 1:10
> sum(x[x>5])
[1] 40
```

Logical vectors may also be turned into index vectors using the function `which()` that returns a vector of all indexes that are `TRUE`.

```
> x <- 1:20
> which(x %% 3 == 0)
[1]  3  6  9 12 15 18
```

Finally, arithmetic functions can also be applied to logical vectors. In this case, logical vectors are coerced to integer vectors with `FALSE` interpreted as `0` and `TRUE` as `1`.

```
> as.integer(c(FALSE, TRUE))
[1] 0 1
> x <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
> y <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
> x + y
[1] 2 1 1 0 1
> sum(x)
[1] 3
```

You can explicitly coerce vectors to type `logical` using the function `as.logcial` that converts 0 to `FALSE`and all other values to `TRUE`.

```
> as.logical(-2:5)
[1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> as.logical(c(0.0, 0.000001, 10000))
[1] FALSE  TRUE  TRUE
```

4.2.1 Exercises: Logical vectors

See Section 18.0.7 for solutions.

1) Given `x <- seq(-20, 20, length.out = 41)` and `y <- seq(-40, 40, length.out = 41)`, calculate the number of pairs(`x[i]`, `y[i]`) where `y[i] > x[i]`.

2) Replace all even numbers of `x` by 1 (hint: use the modulus operator).

3) Extract all elements of `x` that are larger or equal zero and are odd numbers.

4) Extract all elements of `x` that are smaller than -10 or larger than 13.

5) Use a logical vector to replace every fourth element of `x` by 1.

4.3 Character Vectors

Text in R is represented by character vectors. In programming, text often is referred to as a string. We can assign strings to variables, just like we did it before with numbers and bools:

```
> x <- "hello"
> y <- 'world'
```

Strings are entered using either matching double (") or single (') quotes. We've previously discussed that vectors may contain strings or numbers, but only elements of one type. Mixing numbers and strings results in numeric elements being coerced to strings:

```
> c(3.46, TRUE, "hello", 1+6i, 3L)
[1] "3.46"  "TRUE"  "hello" "1+6i"  "3"
```

The function `paste()` is an important function that you can use to create and build strings. `paste()` takes one or more R objects, converts them to type "character", and then concatenates them to form strings.

```
> paste("one", "two", "three")
[1] "one two three"
```

As you can see, the default separator is a blank space (`sep = " "`). You can also select another character, e.g. `sep = "..."`:

```
> paste("one", "two", "three", sep = "...")
[1] "one...two...three"
```

The vectors passed to `paste()` can have length > 1. In this case, they are concatenated term-by-term, and a vector is returned. We can use the argument `collapse` if we want to concatenate the values in the result to a single string.

```
> x <- c("one", "two", "three")
> y <- c("X", "Y", "Z")
> paste(x, y, sep = "...")
[1] "one...X"   "two...Y"   "three...Z"
> paste(x, y, sep = "...", collapse = "-")
[1] "one...X-two...Y-three...Z"
```

If you give `paste()` vectors of different length, then it will apply the recycling rule. This means that it will just re-use the elements of the shorter vector until it reaches the end of the longer vector:

```
> paste(c("X", "Y"), 1:5)
[1] "X 1" "Y 2" "X 3" "Y 4" "X 5"
```

Sometimes, you might want to concatenate the arguments without seperator. In this case, either use `sep = ""` (empty string) or the function `paste0()`:

```
> paste("one", "two", "three", sep = "")
[1] "onetwothree"
> paste0("one", "two", "three")
[1] "onetwothree"
```

4.3.1 Exercises: Character vectors

See Section 18.0.8 for solutions.

1) Create a single string with elements A1, B2, A3, B4 . . . , A99, B100.

2) Create a single string with elements A1, B1, A2, B2, . . . , A100, B100.

3) Given the following vectors `hello <- "Hello"`, `world <- "World"` and `bang <- "!"`, how would you use `paste()` and `paste0()` to get the following strings: `"Hello World !"` `"HelloWorld!"` `"Hello World!"` `"Hello-World!"` `"Hello World!!!"` `"Hello World! Hello World! Hello World!"`?

4.4 Factors

Factors are data objects that are used to group data into categories. These categories are referred to as levels and can be both strings and integers. They are useful if there are limited number of unique values, for example sex, days of the week, geographic clusters etc.

Suppose we have sampled people from different cantons of Switzerland. These are listed in a vector. We can factorize these locations using the function `factor()`:

```
> locations <- c("BE", "FR", "VD","FR", "BE", "FR", "VS", "FR","VD","FR", "BE", "BE", "VD", "FR",
> fLocations <- factor(locations)
> fLocations
 [1] BE FR VD FR BE FR VS FR VD FR BE BE VD FR VD BE BE FR BE
Levels: BE FR VD VS
```

As you see, R detected 4 categories (levels), which are `BE`, `FR`, `VD` and `VS`. Each entry in the vector `fLocations` is not a string anymore, but refers to one of these levels. In fact, factors are stored as integer vectors. This can be seen from its structure:

```
> str(fLocations)
 Factor w/ 4 levels "BE","FR","VD",..: 1 2 3 2 1 2 4 2 3 2 ...
```

We see that levels are stored in a character vector and the individual elements are actually stored as indices.

Factors can easily be used to group elements of other vectors. Say we know the canton as well as the height of every person we sampled:

```
> height <- c(151, 159, 132, 144, 147, 144, 152, 139, 152, 139, 158, 141, 149, 144, 141
> tapply(height, fLocations, mean)
      BE       FR       VD       VS
144.8571 145.1429 143.5000 152.0000
```

Don't worry about the function `tapply()` for the moment, we will discuss it later in more detail. It basically extracts all individuals with factor `BE` from the vector of heights and computes the mean of these individuals. Then it extracts all individuals with factor `FR`, computes the mean etc.

4.4.1 Exercises: Factors

See Section 18.0.9 for solutions.

1) if `d <- c(7, 8, 3, 3, 5, 3, 8, 4, NA)`, what are the levels of `factor(d)`?

2) Create a vector `months` of strings with elements "july", "december", "july", "july", "may", "december", "july", "september", "may", "october", "december", "december", "august", "may", "september", "december", "september". Now factorize it. How many levels are there? What are their names?

4.5 Missing values and such

While we all strive for perfect data, missing data is common in real data sets. In R, missing data is represented by `NA`, meaning not available. `NA` is a special value that can not easily be compared. For instance, we can not say if `NA` is > 0, <0 or $== 0$.

```
> a <- NA
> a < 0
[1] NA
> a == 0
[1] NA
> a > 0
[1] NA
```

In fact, any comparison with `NA` is `NA` itself, as there is nothing that can be said about missing data. Of course, we can not even say if some missing data is equal to other missing data.

```
> NA == NA
[1] NA
```

To test if some data is missing, we thus need to use to function `is.na()`.

```
> is.na(NA)
[1] TRUE
```

Most functions applied to vectors also struggle with `NA` values. What is the average of unknown data? Well, unknown.

```
> x <- c(1,3,7, NA, 9)
> mean(x)
[1] NA
```

In many cases, we may wish to exclude missing values for calculations. This can be done using a logical vector created by `is.na()`, or by using the argument `rm.na` that many functions offer.

```
> x <- c(1,3,7, NA, 9)
> mean(x[!is.na(x)])
[1] 5
> mean(x, na.rm=TRUE)
[1] 5
```

Apart from `NA`, R knows a bunch of other special values, including `NaN` (not a number) or `Inf` and `-Inf` denoting ∞ and $-\infty$, respectively. They mostly arise through calculations and usually inform us that we do something fishy...

```
> 1/0
[1] Inf
> 0/0
[1] NaN
> log(-1)
Warning in log(-1): NaNs produced
[1] NaN
```

4.6 Sorting, Shuffling and Sampling Vectors

The function `sort()` returns a sorted version of a vector:

```
> x <- c(-8, 5, 8, 2, 47, 10, -3, 2, 218, -27, 72, -73, 3)
> sort(x)
 [1] -73 -27  -8  -3   2   2   3   5   8  10  47  72 218
> sort(x, decreasing=T)
 [1] 218  72  47  10   8   5   3   2   2  -3  -8 -27 -73
```

The rank of the elements can be obtained with the optional argument `index.return`, which returns an index vector indicating the correct order:

```
> x <- c(-8, 5, 8, 2, 47, 10, -3, 2, 218, -27, 72, -73, 3)
> sort(x, index.return=T)
$x
 [1] -73 -27  -8  -3   2   2   3   5   8  10  47  72 218

$ix
```

```
 [1] 12 10  1  7  4  8 13  2  3  6  5 11  9
> x[sort(x, index.return=T)$ix]
 [1] -73 -27  -8  -3   2   2   3   5   8  10  47  72 218
```

We can use this index vector to sort e.g. `x` (in this case, we get exactly the same as `sort(x)`, see above). However, the index vector can be very useful if we want to sort a vector `y` according to another vector `x`:

```
> x <- c(-8, 5, 8, 2, 47, 10, -3, 2, 218, -27, 72, -73, 3)
> y <- c(7, 28, 49, 1, -28, 2, 49, 12, 49, 5, -1, 1, 2)
> y[sort(x, index.return=T)$ix]
 [1]   1   5   7  49   1  12   2  28  49   2 -28  -1  49
```

The opposite of sorting is shuffling, in which case the elements of a vector are, well, shuffled at random. This can be achieved using the function `sample()`:

```
> x <- 1:10
> sample(x)
 [1]  2  4  7  1  3  9 10  5  6  8
```

While we used the function `sample()` to shuffle a vector, the function can also be used what its name suggests: to sample elements from a vector. For this, `sample()` has the argument `size` that indicates the number of elements to choose. By default, this argument is equal to the length of the provided vector - but we can use it to generate sub-samples easily.

```
> sample(-5:5, size=3)
[1] -1 -2  2
```

Obviously, we can not take more samples than elements provided, unless we sample with replacement.

```
> sample(c(10,20), size=10)
Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than t
> sample(c(10,20), size=10, replace=TRUE)
 [1] 10 20 10 10 20 10 20 10 10 10
```

By default, each element has the same probability to be sampled. However, you may provide specific probabilities using the argument `prob`. These probabilities will automatically be normalized to so they sum to 1.

```
> sample(1:2, prob=c(0.01, 0.99))
[1] 2 1
> sample(1:2, prob=c(100,1))
[1] 1 2
```

4.6.1 Exercises: Sorting and shuffling vectors

See Section 18.0.10 for solutions.

1) Create a vector `x` with elements 3, 4, -6, 2, -7, 2, -1, 0 and sort it in increasing and decreasing order.

2) Take the same vector `x` with elements 3, 4, -6, 2, -7, 2, -1, 0, but this time only sort the first three elements in increasing order.

3) Create a vector `v` with 98 evenly spaced elements between 8 and 37. Then from `v`, sample 10 elements at random and store these 10 elements in a new vector called `V`. Then, sort `V` in decreasing order.

Chapter 5

Matrices

Matrices are the extention of a vector to two dimensions that can only contain one data class and frequently used in statistics. If you are unfamiliar with matrix operations, don't worry: this chapter introduces how to create and use matrices in R.

5.1 Creating matrices

Matrices are created with the function `matrix()`:

```
> matrix(1:12, nrow=3, ncol=4)
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Note that in mathematics, matrices are usually denoted by a (bold) captial letter, and we will adopt this convention here.

The example above creates a 3x4 matrix with 3 rows and 4 columns, filled with the numbers from 1 through 12. Notice that the function `matrix()` takes a vector as input and fills it into the matrix of the specified size. By default, the matrix is filled column-by-column, but it can also be filled row-by-row by setting the argument `byrow=TRUE`.

```
> matrix(1:12, nrow=3, ncol=4, byrow=TRUE)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Note that in case you provide the function `matrix()` with a vector, there is no

need to specify both dimensions: if a matrix with 3 rows is to be constructed from 12 elements, there must be 4 columns. Hence we can use

```
> matrix(1:12, nrow=3)
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> matrix(1:12, ncol=4)
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

If you provide both dimensions and the length of the provided vector does not match, it will either be subset or recycled.

```
> matrix(1:12, nrow=2, ncol=3)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix(1:3, nrow=2, ncol=3)
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    2    1    3
> matrix(1, nrow=2, ncol=3)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
```

Just as for vectors, R prints a warning if the provided vector is not a multiple or submultiple of the number of required elements.

```
> matrix(1:10, nrow=2, ncol=3)
Warning in matrix(1:10, nrow = 2, ncol = 3): data length [10] is not a sub-
multiple or multiple of the number of columns [3]
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

To check for the dimensionality of a matrix, you may use the function `dim()` that returns a vector of two elements, the first being the number of rows and the second the number of columns of the matrix. To directly get the number of rows or columns, use `nrow()`and `ncol()`.

```
> A <- matrix(1:12, nrow=4)
> dim(A)
[1] 4 3
```

```
> nrow(A)
[1] 4
> ncol(A)
[1] 3
```

An important type of matrices in statistics are diagnoal matrices that only have non-zero values on the diagonal. You can generate such matrices easily with the function `diag()`.

```
> diag(1, nrow=4)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
> diag(1:3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

Note that the function `diag()` can also be used to access the diagnoal of a matrix.

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

Since a matrix is a vector to two dimensions, we can also create matrices by combining vectors with the following functions:

`cbind()` bind by column `rbind()` bind by row

This way, each vector would be one column or row (depends on the function used).

```
> vector1 <- 1:8
> vector2 <- 9:16
> vector3 <- 17:24
```

By row

```
> rbind(vector1,vector2,vector3)
        [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
vector1    1    2    3    4    5    6    7    8
vector2    9   10   11   12   13   14   15   16
vector3   17   18   19   20   21   22   23   24
```

By column

```
> cbind(vector1,vector2,vector3)
     vector1 vector2 vector3
[1,]       1       9      17
[2,]       2      10      18
[3,]       3      11      19
[4,]       4      12      20
[5,]       5      13      21
[6,]       6      14      22
[7,]       7      15      23
[8,]       8      16      24
```

Finally, matrices can contain NAs.. for instance:

```
> vector1 <- c(NA, NA, 3)
> vector2 <- c(NA,  5,  NA)
>
> rbind(vector1, vector2)
        [,1] [,2] [,3]
vector1   NA   NA    3
vector2   NA    5   NA
```

5.1.1 Exercises: Creating matrices

See Section 18.0.11 for solutions.

1. Create a 5x10 matrix filled with 1.
2. Create a diagonal 10x10 matrix with elements 1,2,1,2, ...
3. Create a matrix `A` with the two rows 1, 2, ..., 10 and 10, 9, ..., 1.
4. Create the same matrix A but using rbind().
5. Generate a vector with the numbers 1, 2, ... 80 and assign it to the variable `b`. Now turn this vector into a matrix `B` with 5 columns. Finally, assign the number of rows of `B` to variable `k`.
6. Generate a 3x3 matrix with all diagonal elements being 1 and all other elements being 2.

5.2 Accessing Elements

Just as for vectors, elements of a matrix are accessed with the `[]` operator, but each element is specified by two indices: one for the row and one for the column.

```
> B <- matrix(1:12, ncol=4)
> B[2,3]
[1] 8
```

These operators can also be used to access (or slice) any subset of a matrix using index or locigal vectors. To just get the second column of the first two rows, for instance, you can use

```
> B[1:2,2]
[1] 4 5
```

Or you may want to get the second and fourth column of all but the second row:

```
> B[-2,c(2,4)]
     [,1] [,2]
[1,]    4   10
[2,]    6   12
```

Based on the slice, R will either return a matrix or a vector.

Sometimes, you may want to access an entire row or an entire column. This is readily done by leaving an index empty, which implies using the full range of possible indexes.

```
> B[2,]
[1]  2  5  8 11
> B[,3]
[1] 7 8 9
```

Just as for vectors, matrix elements can also be accesed using logical matrices (or vectors). A logical matrix can be created either with `matrix()`

```
> matrix(c(TRUE, TRUE, FALSE), nrow=3, ncol=2)
      [,1]  [,2]
[1,]  TRUE  TRUE
[2,]  TRUE  TRUE
[3,] FALSE FALSE
```

or using conditions such as

```
> matrix(1:12, nrow=3) %% 2 == 0
      [,1]  [,2]  [,3]  [,4]
[1,] FALSE  TRUE FALSE  TRUE
[2,]  TRUE FALSE  TRUE FALSE
[3,] FALSE  TRUE FALSE  TRUE
```

This allows us to for instance replace all odd elements with 0

```
> A <- matrix(1:12, nrow=3);
> A
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> A[A %% 2 == 1] <- 0
> A
     [,1] [,2] [,3] [,4]
[1,]    0    4    0   10
[2,]    2    0    8    0
[3,]    0    6    0   12
```

Finally, elements of a matrix can also be accessed via the `[]` operator using a single index. In this case, the index is interpreted as linearized into a vector by stacking the columns.

```
> A <- matrix(1:12, nrow=3, byrow=FALSE)
> A
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> A[c(5,7)]
[1] 5 7
> A <- matrix(1:12, nrow=3, byrow=TRUE)
> A
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> A[c(5,7)]
[1] 6 3
```

This then also allows us to use the function `which()` to assign values based on a vector of indexes that pass a given condition.

```
> A <- matrix(-2:2, nrow=4, ncol=5)
> which(A < 0)
[1]  1  2  6  7 11 12 16 17
> A[which(A<0)] <- 420
> A
     [,1] [,2] [,3] [,4] [,5]
[1,]  420    2    1    0  420
[2,]  420  420    2    1    0
[3,]    0  420  420    2    1
```

```
[4,]    1    0  420  420    2
```

5.2.1 Exercises: Accessing Matrix Elements

See Section 18.0.12 for solutions.

1. Create a 4x4 matrix A with elements 1, 2, ..., 16 filled row by row (first row is 1, 2, 3, 4). Assign a slice of all but the last row to a new matrix B.
2. What is the sum of all elements in the second row of the matrix A from above? And of all elements of the third column?
3. Replace all elements A_{ij} of matrix $\boldsymbol{A}$ for which $A_{ij} + 1 < \left(\frac{A_{ij}}{3}\right)^2$ with value -1. **tip: which(condition)**

5.3 Matrix Arithmetics

Just as with vectors, most arithmetic operations in R are applied to individual elements of a matrix. That applies to the standard arithmetic operations as well as to most functions.

```
> A <- matrix(c(0, -1, 1, 0), nrow=2)
> A
     [,1] [,2]
[1,]    0    1
[2,]   -1    0
> A*2
     [,1] [,2]
[1,]    0    2
[2,]   -2    0
> A/7
           [,1]      [,2]
[1,]  0.0000000 0.1428571
[2,] -0.1428571 0.0000000
> log(A)
Warning in log(A): NaNs produced
     [,1] [,2]
[1,] -Inf    0
[2,]  NaN -Inf
```

Also just as with vectors, some functions are applied to the matrix as a whole.

```
> A <- matrix(c(0, -1, 1, 0), nrow=2)
> sum(A)
[1] 0
> mean(A)
[1] 0
```

```
> sd(A)
[1] 0.8164966
```

However, R also supports the matrix product using the operator %*%.

```
> A <- matrix(1:4, nrow=2)
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> B <- matrix(4:1, nrow=2);
> B
     [,1] [,2]
[1,]    4    2
[2,]    3    1
> C <- A %*% B
> C
     [,1] [,2]
[1,]   13    5
[2,]   20    8
```

Of course, R will throw an error if the matrices are not conformable (i.e. if their dimensionality prohibits multiplication)

```
> matrix(1, nrow=3, ncol=3) %*% matrix(1, ncol=1, nrow=3)
     [,1]
[1,]    3
[2,]    3
[3,]    3
> matrix(1, nrow=3, ncol=3) %*% matrix(1, ncol=2, nrow=1)
Error in matrix(1, nrow = 3, ncol = 3) %*% matrix(1, ncol = 2, nrow = 1): non-conformal
```

R can also perform other standard matrix operations such as transposing or inverting a matrix. The transpose $\boldsymbol{A}^T$ of a matrix $\boldsymbol{A}$ (the swapping of rows and columns) is done using the function `t()`.

```
> D <- matrix(1:9, nrow=3)
> D
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> t(D)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

The inverse operation to a matrix multiplication. To invert a matrix, use the function `solve()`. Using the matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ from above, we can show that $\boldsymbol{C} \times \boldsymbol{B}^{-1} = \boldsymbol{A}$.

```
> C %*% solve(B)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

If you are unfamiliar with transposing, mulitplying or inverting matrices, don't worry about it - but it does not hurt to know that they exists and R can perform them!

5.3.1 Exercises: Matrix Arithmetic

See Section 18.0.13 for solutions.

1. Consider a matrix $\boldsymbol{Z} = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}$, what is the matrix product $\boldsymbol{Z} \times \boldsymbol{Z}^T$?
2. What is the matrix product $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} -1 & 1 & -1 \\ 1 & -1 & 1 \end{pmatrix}$?
3. What is the inverse of the matrix $\begin{pmatrix} 1 & -3 & 7 \\ -2 & 0 & 1 \\ -1 & 1 & 0 \end{pmatrix}$?
4. Create two different 2 by 2 matrices named E and G. E should contain the values 1 to 4 and G the values 5 to 8. Try out the following commands and by looking at the results see if you can figure out what is going on.
   - $\boldsymbol{E} * \boldsymbol{G}$ (multiplication *)
   - $\frac{E}{G}$
   - $\boldsymbol{E} \times \boldsymbol{G}$ (matrix multiplication %*%)
   - $\boldsymbol{E} + \boldsymbol{G}$
   - $\boldsymbol{E} - \boldsymbol{G}$
   - $\boldsymbol{E} == \boldsymbol{G}$

Chapter 6

Data Frames

A data frame is basically a tidy spreadsheet: a bunch of data organized in a bunch of columns. From an R point of view, a data frame is a collection of vectors of equal length, corresponding to the columns of our imaginary spreadsheet. However, and in contrast to matrices, the vectors can differ in their type and be either numeric, character or logical.

6.1 Creating data frames

Data frames are constructed using the function `data.frame()`, which takes the vectors of the data frame as arguments.

```
> x <- c(1,-7,10,12,6,8,6,5,-5,-10);
> d <- data.frame(x=x, negative=x<0, 1:10);
> d
     x negative X1.10
1    1    FALSE     1
2   -7     TRUE     2
3   10    FALSE     3
4   12    FALSE     4
5    6    FALSE     5
6    8    FALSE     6
7    6    FALSE     7
8    5    FALSE     8
9   -5     TRUE     9
10 -10     TRUE    10
```

If names are provided to `data.frame()`, they are used as column names. Otherwise, R creates a unique name automatically, as is the case for the third column in the example above. That example also illustrates that vectors can either be provided as variables (e.g. `x`) or created on the fly (e.g. `x<0`).

You may add additional columns to a data frame using the function `cbind()` that "binds" columns.

```
> a <- 1:10; b <- rep(c("x", "y"), 5);
> d <- cbind(a, b);
> d <- cbind(d, a<5);
> d
      a    b
 [1,] "1"  "x" "TRUE"
 [2,] "2"  "y" "TRUE"
 [3,] "3"  "x" "TRUE"
 [4,] "4"  "y" "TRUE"
 [5,] "5"  "x" "FALSE"
 [6,] "6"  "y" "FALSE"
 [7,] "7"  "x" "FALSE"
 [8,] "8"  "y" "FALSE"
 [9,] "9"  "x" "FALSE"
[10,] "10" "y" "FALSE"
```

On the other hand, the function `rbind()`returns a data frame by binding rows.

```
> e <- data.frame(word=c("one", "two"), number=c(1, 2));
> f <- data.frame(word=c("three", "four","five","six"), number=3:6);
> d <- rbind(e, f);
> d <- rbind(d,data.frame(word=c("seven"), number=7));
> d
   word number
1   one      1
2   two      2
3 three      3
4  four      4
5  five      5
6   six      6
7 seven      7
```

When using `rbind()`, the data frames should have the same column names, otherwise merging the data frames produces an error. You can, however, use `names()` to check if the column names are the same and rename the column names of one of the data frames to match.

```
> d <- data.frame(words=paste("word", 1:4, sep=""), numbers=c(1,2,3,4))
> a <- data.frame(word=c("word5"), number=c(5))
> names(d) == names(a)
[1] FALSE FALSE
> names(a) <- names(d)
> d <- rbind(d,a)
> d
  words numbers
```

```
1 word1       1
2 word2       2
3 word3       3
4 word4       4
5 word5       5
```

Just as with matrices, the dimensionality of a data frame is checked with the function `dim()`, `nrow()`and `ncol()`.

```
> d <- data.frame(1:5, paste("Test", letters[1:5]))
> dim(d)
[1] 5 2
> nrow(d)
[1] 5
> ncol(d)
[1] 2
```

6.1.1 Exercises: Data Frames

See Section 18.0.14 for solutions.

1. Create two numeric vectors x and y, with numbers from 0 to 5 (x decreasing, y increasing) and merge them to a data frame.
2. Create a new data frame as in the previous exercise but with numbers from 6 to 10 (vector z decreasing, vector w increasing) and merge this new data frame to your data frame from the previous exercise.
3. Add a logical column ("compared") to your previous data frame that states if the first column is bigger than the second.

6.2 Accessing data frames

Individual vectors of a data frame can easily be accessed using the operator `$`.

```
> f <- data.frame(word=c("one", "two", "three", "four","five","six"), number=1:6, bigger2=1:6>2);
> f$word
[1] "one"   "two"   "three" "four"  "five"  "six"
```

Elements of these vectors are then accessed using the `[]` operator, just as on ordinary vectors.

```
> f <- data.frame(word=c("one", "two", "three", "four","five","six"), number=1:6, bigger2=1:6>2);
> f$bigger2[3]
[1] TRUE
> f$number[3]
[1] 3
```

```
> f$word[1:2]
[1] "one" "two"
```

As with matrices, the desired elements can be identified using both a row and a column index vector. Leaving one index vector blank returns the whole set.

```
> f <- data.frame(word=c("one", "two", "three", "four","five","six"), number=1:6, bigge
> f[2,]
  word number bigger2
2  two      2   FALSE
> f[,2]
[1] 1 2 3 4 5 6
> f[4:5,-3]
  word number
4 four      4
5 five      5
```

You can also subset data frames based on logical vectors. As with vectors, a useful function is `which()` that returns the index of `TRUE` element (that can be further used to identify the index of a row for example).

```
> f <- data.frame(word=c("one", "two", "three", "four","five","six"), number=1:6, bigge
> a<-which(f$bigger2)
> f[a,]
   word number bigger2
3 three      3    TRUE
4  four      4    TRUE
5  five      5    TRUE
6   six      6    TRUE
```

Data frames can easily be sorted using the function `order()`.

```
> d <- data.frame(a=c(12,11,20,1,1,5),b=c("lion","monkey","snake","elephant","cat","tig
> order(d$a)
[1] 4 5 6 2 1 3
> d[order(d$a),]
   a        b
4  1 elephant
5  1      cat
6  5    tiger
2 11   monkey
1 12     lion
3 20    snake
> d[order(d$a,d$b),]
   a        b
5  1      cat
4  1 elephant
6  5    tiger
```

```
2 11   monkey
1 12     lion
3 20    snake
```

The function `order()` returns the position/rank of the original value. In the example above, `order(d$a)[1]` is 4, because the fourth element of `a` has value 1 and as such the smallest value. `order(d$a)[2]` is 5, because the fifth element of `a` has again value 1 and so on. Although both values are 1, R still orders them, by default just by the position they appear in the vector. If we pass a second argument to `order()`, it will use this second argument to break such ties. Therefore, `order(d$a,d$b)[1]` is now 5, because the "c" of "cat" comes before the "e" of "elephant" in the alphabet, so R knows how to prioritize.

6.2.1 Exercises: Accessing data frames

See Section 18.0.15 for solutions.

1. Create a data frame with a column x from a vector 1:10 and a column y from a vector c(1,10,-2,3,8,-7,2,1,9,4). Access rows that are the same in the both columns of the data frames (using a logical vector).

2. Calculate the mean of the last five rows of the column y after you divide that column with a column x. You can use `length()` to assess the length of a vector.

6.3 Manipulating data frames

When accessing, you can also change the elements of data frames in place.

```
> f <- data.frame(word=c("one", "two", "three", "four","five","six"), number=1:6, bigger2=1:6>2);
> f[2,1] <- "one"
> f$word[3] <- "anotherOne"
> f[2,2] <- 1
> print(f)
        word number bigger2
1        one      1   FALSE
2        one      1   FALSE
3 anotherOne      3    TRUE
4       four      4    TRUE
5       five      5    TRUE
6        six      6    TRUE
```

Notably, you might want to change a class of a column. You can check the class of a column in a data frame with `class()`. The same function can be called to assess a class of any R element.

```
> class(f$word)
[1] "character"
> class(f$number)
[1] "numeric"
```

R allows us to convert classes with the functions like `as.character()`, `as.numeric()`, `as.factor()` and `as.logical()`. For example, we can change a character vector to factors with the function `as.factor()` (note that if you use a R version older than 4.0, character vectors inside a data frame are converted to factors by default). In similar spirit, we can change a numeric vector to a character vector with `as.character()` - and so on!

```
> f$word <- as.factor(f$word)
> class(f$word)
[1] "factor"
> f$number <- as.character(f$number)
> class(f$number)
[1] "character"
```

If you want to see the structure and the classes of all columns at once, you can use the function `str()`.

```
> str(f)
'data.frame':   6 obs. of  3 variables:
 $ word   : Factor w/ 5 levels "anotherOne","five",..: 4 4 1 3 2 5
 $ number : chr  "1" "1" "3" "4" ...
 $ bigger2: logi  FALSE FALSE TRUE TRUE TRUE TRUE
```

6.3.1 Exercises: Manipulating data frames

See Section 18.0.16 for solutions.

1. Create a data frame `data` containing a vector `id` with the labels X1, X2,. . ., X10 (using `paste`), followed by two vectors `x` and `y` both containing 10 numbers sampled from [-10,10] with replacement. Replace all elements of y with y[i] squared.
2. Add a logical column `ok` to `data` that is `TRUE` for all rows with `y>20` and `FALSE`otherwise. Create a data frame `other` that contains both `id` and `y`, but only for those rows with `x>3` and `ok==TRUE`

Chapter 7

Lists

Lists are a generic container that can hold objects of different classes. This makes lists quite handy to store heterogeneous data in R and are the preferred return type of many functions

List are created with the function `list()`.

```
> x <- list(3.1416,"hello world", FALSE, c(1:10))
> x
[[1]]
[1] 3.1416

[[2]]
[1] "hello world"

[[3]]
[1] FALSE

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

The first element is a numeric vector, the second is a character vector, the third is a logical vector and the fourth an integer vector. As we can see, containing different classes is not a problem with lists.

However, the list is displayed a little bit different. It doesn't print out like a vector because every element is different.

7.1 Subsetting a list

Just as with vectors and matrices, individual elements of a list can be accessed using the `[]` operator. However, the single square brackets operator `[]` always

returns an element that is of the same class as the original. In the case of lists, the operator `[]` will thus return a list as well.

```
> x <- list(3.1416,"hello world", FALSE, c(1:10))
> print(x[1])
[[1]]
[1] 3.1416
> class(x[1])
[1] "list"
```

While this is handy when working with lists, we might sometimes also access an element of a list in its own type, we need to use the double backet operator `[[]]`.

```
> x <- list(3.1416,"hello world", FALSE, c(1:10))
> class(x[1])
[1] "list"
> class(x[[1]])
[1] "numeric"
> class(x[[2]])
[1] "character"
> class(x[[3]])
[1] "logical"
> class(x[[4]])
[1] "integer"
```

Elements of list might be named just as columns of a data frame. If they are named, they can also be accessed with the dollar sign operator `$`. That is quite practical because we don't need to remember the order of elements in the list.

```
> y <- list(food=c("pizza","tacos"), price=c(14,25))
> class(y)
[1] "list"
> y$food
[1] "pizza" "tacos"
> class(y$food)
[1] "character"
> y$price
[1] 14 25
> class(y$price)
[1] "numeric"
```

The name can further be passed to the `[[]]` operator instead of a numeric index.

```
> y <- list(food=c("pizza","tacos"), price=c(14,25))
> y[['food']]
[1] "pizza" "tacos"
> y[['price']]
```

```
[1] 14 25
```

However, there are also some nice use cases for the single square bracket operator. Suppose we want to extract not one but two elements of a list, then we have to use `[]`. For instance:

```
> y <- list(food=c("pizza","tacos"), price=c(14,25), available=c(TRUE,FALSE))
> y[c(1,3)]
$food
[1] "pizza" "tacos"

$available
[1]  TRUE FALSE
```

In the example above, a numeric vector c(1,3) is passed to the list using the single bracket operator and that gives back a list with food and available elements. Extracting multiple elements of a list is not possible wth the $or `[]` operator.

A final note on these operators. Suppose you want to extract an element of a list by its name, but you want this name to be stored in a variables. This is only possible using the `[[]]` operator but not with the `$` operator that only takes literal names.

```
> y <- list(food=c("pizza","tacos"))
> element <- "food"
> y$element
NULL
> y[[element]]
[1] "pizza" "tacos"
```

7.2 Nested elements of a list

For the case where a list contains an element with several values, we have to apply the same logic of how to get an element from a list and then subsetting that element to get the value we want. As an example, consider the case of a list containing a numeric vector. We can use the `[[]]` operator on the list to access that vector, and then the `[]` operator to access an element of the vector.

```
> x <- list(seq(0.0, 1.0, length.out=11))
> x[[1]]
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> x[[1]][3]
[1] 0.2
```

This gets even more beautiful if we access lists within lists.

```
> x <- list(FALSE, list(list(1:5, 10:20), pi))
> x
```

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

[[2]]
[[2]][[1]]
[[2]][[1]][[1]]
[1] 1 2 3 4 5

[[2]][[1]][[2]]
 [1] 10 11 12 13 14 15 16 17 18 19 20


[[2]][[2]]
[1] 3.141593
> x[[2]][[1]][[1]][3]
[1] 3
```

As this may become unreadable, R offers a short cut: the `[[]]` can also be provided a vector of indexes to follow. Revisiting the above eample, we can use the sequence of indexes `c(2,1,1,3)` to access the same element.

```
> x <- list(FALSE, list(list(1:5, 10:20), pi))
> x[[c(2,1,1,3)]]
[1] 3
```

7.3 Lists as return type

Given the flexibility of lists, they are the preferred return type of functions returning multiple values. An example is the function `sort()` that returns a list when using the argument `index.return=T`.

```
> z <- c(5,4,3,6,8,1,9,2,7)
> class(z)
[1] "numeric"
> class(sort(z))
[1] "numeric"
> class(sort(z, index.return=T))
[1] "list"
```

So in case you have wondered in Section 4 (Vectors) why we accessed the rank of sorted elements using the `$` operator, you now know it was because `sort()` returns a named list.

Another common function that returns a list is `strsplit(x,split,...)` that split strings into substrings based on a provided character vector.

```
> x <- "Chapter 7. Lists"
> strsplit(x,"\\.")
[[1]]
[1] "Chapter 7" " Lists"
> strsplit(x,"\\.")[[c(1,2)]]
[1] " Lists"
> strsplit(x," ")
[[1]]
[1] "Chapter" "7."       "Lists"
> strsplit(x," ")[[c(1,3)]]
[1] "Lists"
```

In the example above, there are two substrings that can be splitted by using the character "." (\ needed to double escape special characters), and three substrings with the space character.

7.3.1 Exercises: Lists

See Section 18.0.17 for solutions.

1. Create a list "holiday" with two character vectors: the first named country with containing "CH", "USA", "Japan", the second named continent containing "Europe", "The Americas", "East Asia".
2. Create a list with four four elements. 1) the list "holiday" from the exercise above, 2) a character vector "place" containing "mountain" and "beach", 3) a numerical vector "day" containing the numbers 1.0 through 20.0, and 4) a logical vector "like" containing `TRUE`and `FALSE`.
3. Return the single element "USA" from the previous list using twice `[[]]` and once `[]`
4. Return the single character "Europe" using each operator `$`, `[[]]` and `[]` just once
5. Return the single character "Japan" using only the operator `[[]]` and only once.
6. Create a variable "name" and set it to "day". Use it to extract the element "day" from the list "x".
7. Subset the list into a new list containing the "place" and "like" elements.
8. Create a chracter vector "Chapters" and concatenate 4 strings. 1) first "Chapter 3. R-Scripts" 2) second "Chapter 4. Vectors" 3) "Chapter 5. Matrices" 4) "Chapter 6. Data Frames". Extract just the word "Data Frames" with no spaces as a character from Chapters.

Chapter 8

Conditionals and Loops

So far, each statement in our code was executed just once. The following two concepts, conditionals and loops, are structures which allow you to repeat or alter the sequence in which the code is executed.

This creates more complex behavior and more interesting functionality.

8.1 Conditionals

The R language offers the conditional construction `if(expr){ ... }` construction where `expr` is a logical expression. If the expression is `TRUE`, the code inside the curly brackets denoted here by `...` gets executed, else it is not.

```
> if(TRUE){ print("yes") }
[1] "yes"
> if(FALSE){ print("yes") }
```

The expression can be arbitrarily complicated, but must evaluate to a single logical value.

```
> if(1+2 < 5*20){ print("yes") }
[1] "yes"
```

The conditional construction can be extended to allow perform an alternative function if the condition is not passed with `if(expr){ ... }else{ ... }`.

```
> x <- 5
> if(3>x){print("yes")} else {print("no")}
[1] "no"
>
> if(sum(1:x)>10 && x*x < x^x){
+   x <- sqrt(x)
```

```
+   print(paste("x is now", x))
+ } else {
+   print(paste("x remains", x))
+ }
[1] "x is now 2.23606797749979"
```

Obviosuly, conditional can be nested.

```
> x <- 52;
> if(x < 0){
+   print("x is negative");
+ } else {
+   if(x %% 2 == 0){
+     print("x is positive and even");
+   } else {
+     print("x is positive and odd");
+   }
+ }
[1] "x is positive and even"
```

Sometimes, including in the case above, a nested construct can be written more elegantly using the `if(){ ... } else if(){ ... }` construct.

```
> x <- 52;
> if(x < 0){
+   print("x is negative");
+ } else if(x %% 2 == 0){
+   print("x is positive and even");
+ } else {
+   print("x is positive and odd");
+ }
[1] "x is positive and even"
```

8.1.1 Exercises: Conditionals

See Section 18.0.18 for solutions.

1. Use conditionals to that informs you if the elements of a vector x of length 3 are strictly increasing, or not, by pritting an appropriate message. Test it on the vector `x <- c(1,4,3)`.

8.2 Loops

Sometimes, a piece of code is meant to be executed multiple times. For this, R offers multiple looping constructs, including `for` and `while`.

8.2.1 The `for` Loop

The most important is the `for` loop, which executes a series of commands for each element of an object such as a vector or a list. The basic synthax of the for loop is `for(i in obj){ ... }`, where the code `...` gets executed for each element in the object `obj`. In each iteration of the loop, the current element of `obj` is available through the variable `i`. Note that the name of the variable can be freely chosen.

```
> for(x in 1:10){ print(x) }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
> b <- c("one","two","three")
> for(j in b){ print(j) }
[1] "one"
[1] "two"
[1] "three"
```

Sometimes it may be more helpful to loop over an object using indexes. This is commonly done by creating a vector of indexes using `seq()` or the `:` operator. In such cases it is also advisable to not specify the length of the indexes explictely, but to use `length()` or `dim()` to obtain the appropriate length. This makes the code easier to maintain.

```
> d <- data.frame(a=1:10, b=seq(0, 1, length.out=10))
> for(i in 1:dim(d)[1]){
+   print(d[i,]);
+ }
  a b
1 1 0
  a         b
2 2 0.1111111
  a         b
3 3 0.2222222
  a         b
4 4 0.3333333
  a         b
5 5 0.4444444
  a         b
```

```
6 6 0.5555556
  a         b
7 7 0.6666667
  a         b
8 8 0.7777778
  a         b
9 9 0.8888889
    a b
10 10 1
```

The code inside a `for` loop can be arbitrarily complex, including other `for` loops or conditionals.

```
> for(i in 1:3){
+   for(j in i:4){
+     if(i %% 2 == 0){
+       print(paste(i, "+", j, "=", i+j))
+     } else {
+       print(paste(i, "*", j, "=", i*j))
+     }
+   }
+ }
[1] "1 * 1 = 1"
[1] "1 * 2 = 2"
[1] "1 * 3 = 3"
[1] "1 * 4 = 4"
[1] "2 + 2 = 4"
[1] "2 + 3 = 5"
[1] "2 + 4 = 6"
[1] "3 * 3 = 9"
[1] "3 * 4 = 12"
```

Nesting `for` loops is particularly useful when looping over two-dimensional objects such as matrices.

```
> a <- matrix(c(1,2,3, 11,12,13), nrow = 2, ncol = 3, byrow = TRUE)
> for(i in 1:nrow(a)){
+   for(j in 1:ncol(a)){
+     print(paste("The element in the row",i, "and in the column", j, "is:", a[i,j]))
+   }
+ }
[1] "The element in the row 1 and in the column 1 is: 1"
[1] "The element in the row 1 and in the column 2 is: 2"
[1] "The element in the row 1 and in the column 3 is: 3"
[1] "The element in the row 2 and in the column 1 is: 11"
[1] "The element in the row 2 and in the column 2 is: 12"
[1] "The element in the row 2 and in the column 3 is: 13"
```

8.2.2 The `while` Loop

The `while` loop combined the functionalities of loops and conditionals. It has the basic synthax `while(expr){ ... }` and executes the code `...` as long as `expr` evaluates to `TRUE`.

```
> x <- 0;
> while(x < 5){
+   print(x)
+   x <- x + 1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
```

It is relatively easy to create infinite loops if the condition is always `TRUE`. An example would be the following (be careful in trying it out!).

```
> x <- 1;
> while(x > 0){
+   print(x)
+   x <- x + 1
+ }
```

Should you end up in an infinite loop, you will need to manually abort R. In RStudio, you can do this via the red **STOP** button in the console panel. If you run R on the command line, hit `CTRL`+`C`.

Note that most loops can be written both as a `for` or `while` loop and which one to use is a matter of choice.

8.2.3 Exercises: Loops

See Section 18.0.19 for solutions.

1. Use a `for` loop to test where a vector of arbitrary length is strictly increasing. If it is not, it should print "Vector is not strictly increasing!".
2. Use two `for` loops to create a vector containing the elements 1, 2, 2, 3, 3, 3, ...
3. Create a matrix with 3 rows containgin the numbers 1, 2, ..., 120. Loop over each column, calculate the sum of the values inside this column and store them in a new vector.
4. Replicate the call `paste("I like number", 1:10)` using a `while` loop.
5. Use a loop to create the first 100 elements of the (https://en.wikipedia.org/wiki/Fibonacci_number)[Fibonacci sequence] 0, 1, 1, 2, 3, 5, ... The

Fibonacci sequence starts with 0 and 1, and each of the following elements $F_n = F_{n-1} + F_{n-2}$.

6. Create two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ containg both the numbers 1:100 in shuffled order. Then loop over these vectors and print the prodcut whenever $x_i > y_i$ and the sum when $x_i < y_i$.

7. Write a loop that prints out the elements of `c(5,6,8,2,3,7,8,1,2,3,4)` until the first number that is smaller than 2.

8. Use nested loops to create the same matrix you get with `matrix(c(2,3,4,3,4,5,4,5,6), nrow = 3, ncol = 3, byrow = TRUE)`.

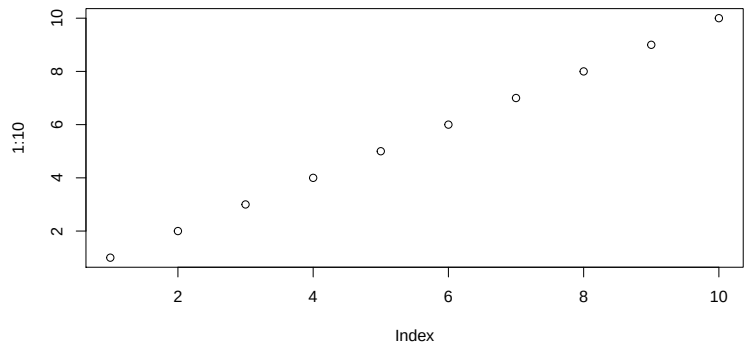
9. Use a loop to investigate the number of iterations I until the product $\prod_{i=1}^{I} i$ reaches 5 million.

Chapter 9

Basic Plotting

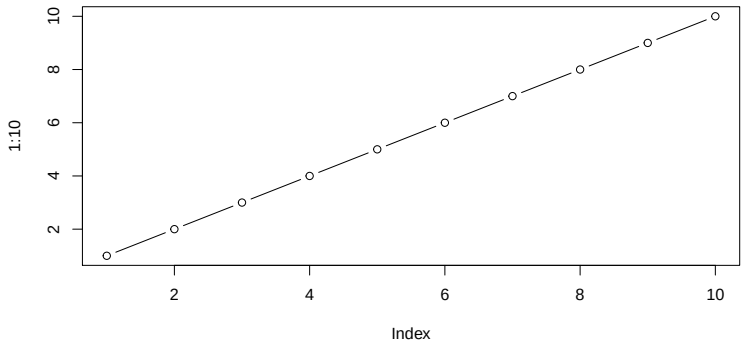
R is particularly powerful in plotting graphs and figures. The generic function to use is `plot()` which gives you a graphical view of the relationship between your data points.

```
> plot(1:10)
```



The `plot()` function takes many default arguments. One of the most important arguments is the way your data is to be plotted: you either indicate `type='l'` for lines, or `type='p'` for points or `type='b'` for both of them. As seen above, the default is `p`.

```
> plot(1:10, type='b')
```

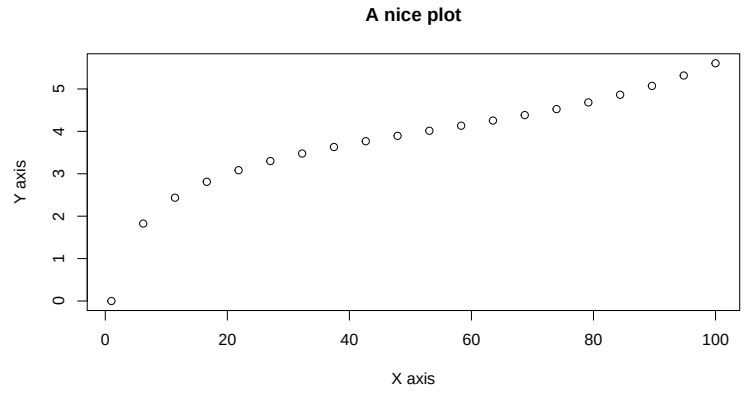


There are many more arguments that you can use to change the appearance of your plot. Let's look at a few of them using the following two example vectors:

```
> x <- seq(1, 100, length.out=20)
> y <- log(x) + (x/100)^5
```

The first things that you usually want to do is to give your plot a title and to properly annotate the axis. The former is achieved with the argument `main`, the latter with the arguments `xlab` and `ylab` that label the x-axis and y-axis, respectively. The default axis labels are the names of your variables, and they are often not very informative.

```
> plot(x, y, xlab='X axis', ylab='Y axis', main='A nice plot')
```



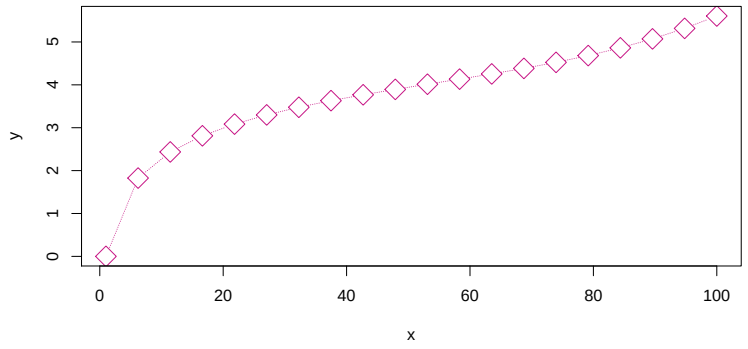
The following arguments are used to change the appearance of the plotted data:

- `col` defines the color. You can either provide the name of a built-in color (you can get all available colors using the function `colors()`), or the numbers 1-8 that are matched to basic colors.
- `pch` defines plotting characters. The numbers 0-25 indicate different symbols, but you can also use any character.
- `cex` (for **character expansion**) defines the size of your plotting characters. Default is `cex=1` and larger values increase the size of the plotting character.

- `lwd` defines the line-width, i.e. how think lines are to be drawn. The default is `lwd=1`.
- `lty` defines the line type. you can either provide a number between 0-6, with `0` being blank (i.e. no line), `1` solid, `2` dashed and so forth, or a combination of two characters from 1, 2, ..., 9, A, ..., F to indicate the length of dashes and spaces. `lty=1F`, for instance, would result in very short dashes with lots of space between them.

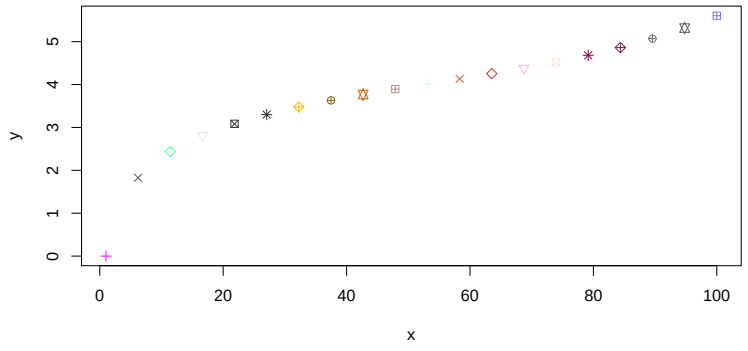
Here is an example of how to use these options.

```
> plot(x, y, type='b', col="mediumvioletred", pch=5, cex=2, lwd=0.5, lty=3)
```



Importantly, all these options can also provided as vectors with one element per data point. If the vectors are shorter, they get recycled.

```
> randomColors <- colors()[sample(1:length(colors()), size=length(x))];
> plot(x, y, col=randomColors, pch=3:12)
```

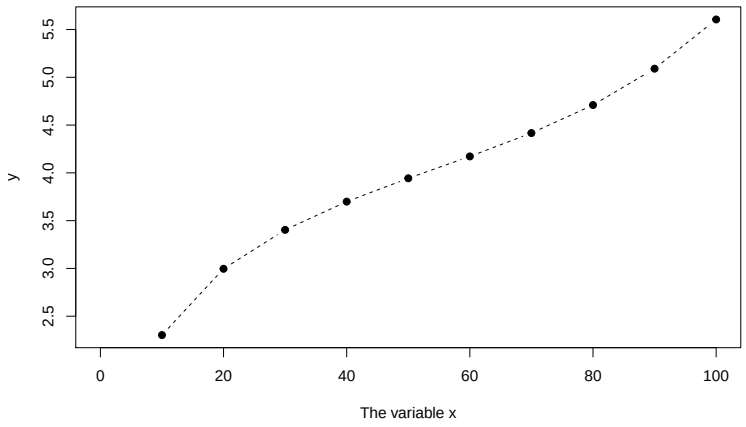


Generally, if you don't know how to plot something, the `?plot` help function is a good place to start, and a quick google search get you usually even faster to a solution.

9.0.1 Permanent Changes: the par() function

Setting graphics parameters with the `par()` function changes the value of the parameters permanently, in the sense that all future calls to graphics functions will be affected by the new value. The `par()` function allows also to set additional graphical parameters including `mar` to set the margins.

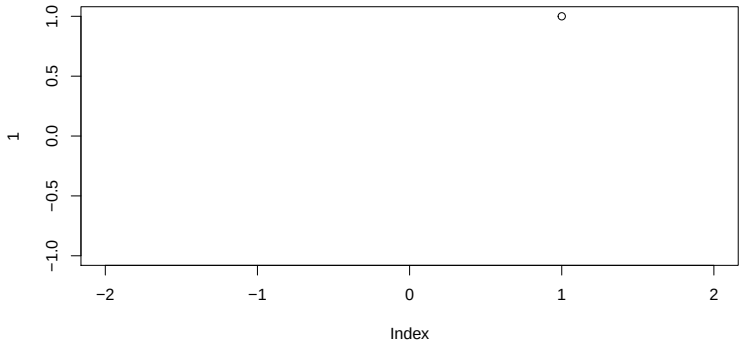
```
> x <- 10*(0:10)
> y <- log(x) + (x/100)^5
> par(mar=c(4,4,0.1,0.1), lty=2, pch=19)
> plot(x,y, type='b', xlab="The variable x")
```



If you check `?par` you will see that the list of options is close to endless and allows to specify pretty much any aspect of your plot.

In addition to specifiying grapical parameters, the `par()` function can also be used to retrieve current settings by providing the name of the graphical parameter of interest. `par("lty")` for instance returns the current line type set. In many cases the call `par("usr")` is particularly useful as it returns a vector of length 4 specifying the coordinate system of the plot: the first pair of values give the x-values at the left-most and right-most position, respectively, and the second pair gives the y-values of the bottom and top of the plot.

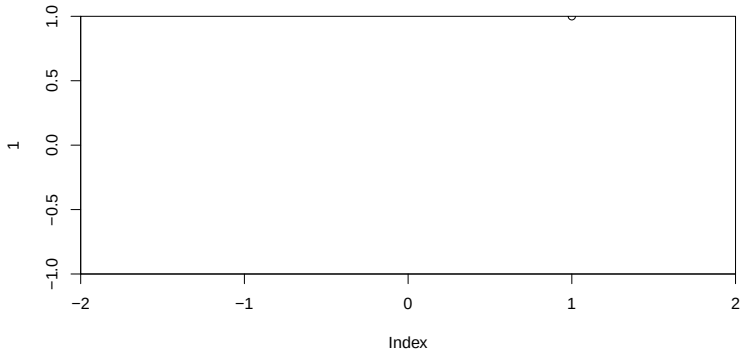
```
> plot(1, xlim=c(-2,2), ylim=c(-1,1))
```




```
> par("usr")
[1] -2.16  2.16 -1.08  1.08
```

As the above example illustrates, R adds some extra space by default. This behavior, while mostly useful, is easily turned off using `xaxs='i'` and `yaxs='i'` for the x and y axis, respectively.

```
> plot(1, xlim=c(-2,2), ylim=c(-1,1), xaxs='i', yaxs='i')
```




```
> par("usr")
[1] -2  2 -1  1
```

9.0.2 Exercises: Basic Plotting

See Section 18.0.20 for solutions.

1. Create a vector `x` of length 20 with equally spaced values between -5 and 5. Then, create a vector `y` with elements $y_i = 2x_i^2$. Plot `y` against `x` using red filled circles connected by dashes. Label the axis "My variable x" and "Some transformation of x" and give the plot the title "The plot of a noob".
2. Plot `y` against `x` again, but this time as a solid, thick line. Use a random color, add no title nor axis-labels and modify the margins to maximizes the area used for the plot itself.
3. In R there are `length(colors())` predefined colors. Make a plot with a

grid of 26x26 points, colored by following the list of available colors. Use large squares as symbols.

4. Plot the numbers 1-100 as filled circles and color them in red if they are a multiple of 7 and in orange other wise.

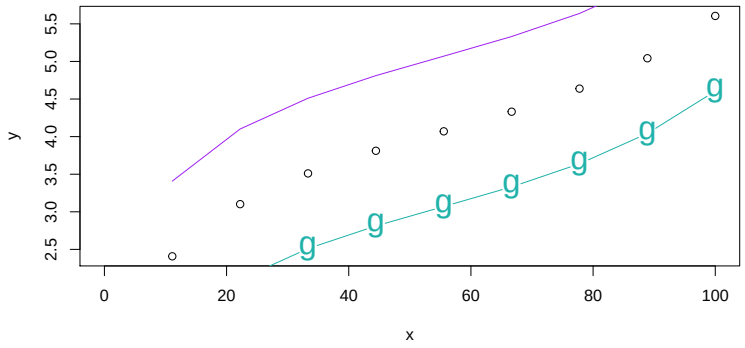
9.1 Multiple Data sets

Often we want to compare multiple data sets in a plot. Here you'll learn the basics of doing so: adding data to an existing plot and providing a legend.

9.1.1 Adding data to a plot

Every time the plot function is called, a new plot is created. To add more data to an already existing plot, you use the commands `points()` or `lines()`. The type arguments can also be provided to those functions. Let's look at an example:

```
> x <- seq(0,100, length.out=10)
> y <- log(x) + (x/100)^5
> plot(x, y)
> lines(x, y+1, col='purple', lwd=0.5)
> points(x, y-1, type='b', pch="g", cex=2, col="lightseagreen")
```



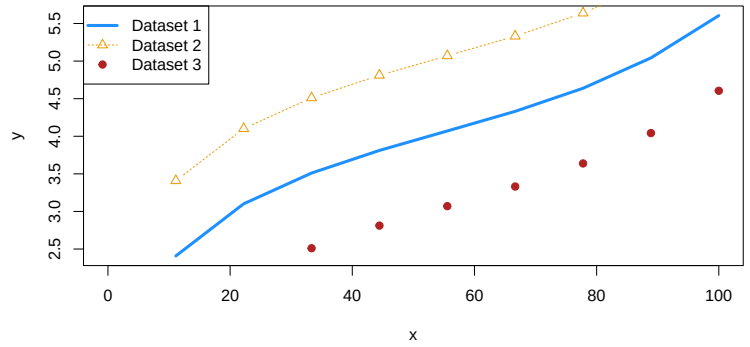
9.1.2 Legends

When plotting something, you might want to add a legend to facilitate the interpretation of your plot. In R, this is done with the `legend()` function. Three sets fo arguments are required:

- The first argument(s) are the position of the legend. You may provide the location to put the legend either using 1) a word such as 'top', 'bottom', 'topleft' and so forth, or 2) the x and y coordinates of the legend.
- The argument `legend` is used to specify the labels to be printed in the legend.

- The formatting to be shown for the different labels. For this, you use the same arguments as for plotting, namely `col`, `lwd`, `lty`, and `pch`, with one element per label entry. Here is an example:

```
> x <- seq(0,100, length.out=10)
> y <- log(x) + (x/100)^5
> plot(x,y, type="l", lwd=3, col="dodgerblue")
> lines(x,y+1, type='b', lwd=0.5, lty=2, pch=2, col="orange2")
> points(x,y-1, pch=19, col="firebrick")
> legend("topleft", legend = c("Dataset 1", "Dataset 2","Dataset 3"), lwd=c(3, 0.5, 1), col=c("do
```



Note that using `lty=NA` and `pch=NA` implies no line or plotting character.

There are many more options that can be used with `legend()`, including `bty` that indicates the type of the box drawn around the legend (use `bty='n'` to have no box) or `horiz` to have the labels organized horizontaly rather than vertically. Check `?legend` for a more comprehensive list. ### Axis

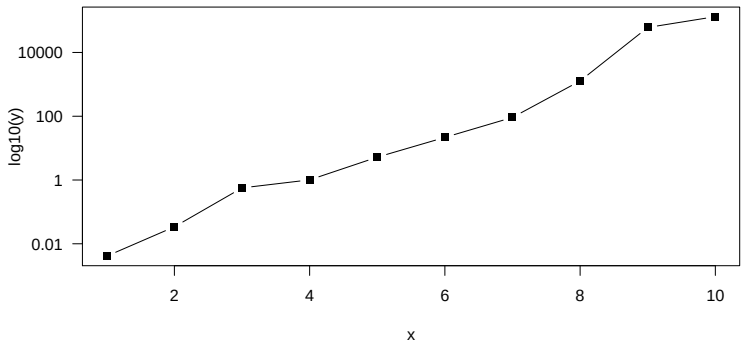
R adds nice axes by default, but sometimes we may want to fiddle with them. Good reasons are ~~deception~~, highlighting specific locations, using multiple axes of opposite sides, or referring to a transformation such as `log()` more intuitively. Axis can be added using the function `axis()` that takes the following main arguments:

- `side` which denotes the site on which to plot the axis. `side=1` refers to the x-axis (bottom), `side=2` to the y-axis (left) and `side=3` and `side=4` to the top and right, respectively.
- `at` denotes the values at which to draw an axis.
- `labels` provides the labels to be printed.

In order to make use of it, you may need to turn default axis printing off. You can do so by setting `xaxt= 'n'` and / or `yaxt='n'`. Here is an exaple:

```
> x <- 1:10
> y <- c(0.0041, 0.034, 0.565, 1.01, 5.2, 21.7, 91.8, 1278, 60756, 132785)
> plot(x, log10(y), yaxt='n', type='b', pch=15)
```

```
> at <- seq(-2, 4, by=2)
> axis(side=2, at=at, 10^at, las=1)
```



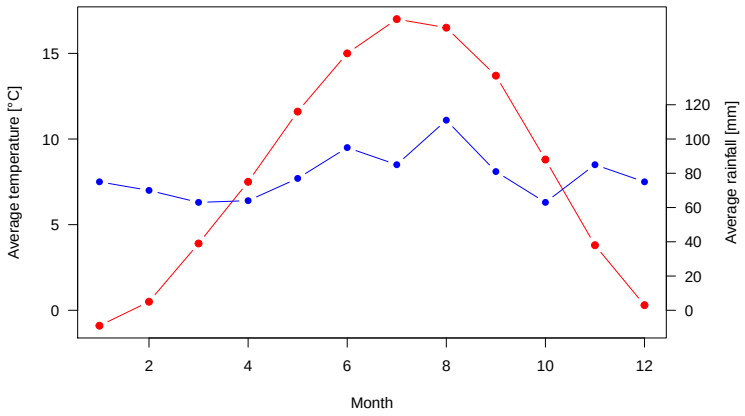
Since choosing nice values for tick marks and labels is an art, R offers the function `pretty()` to produce pretty labels.

```
> pretty(1:17)
[1]  0  5 10 15 20
```

Sometimes we wish to plot multiple data sets in one plot even if they use different units. A classic example are climate diagrams that plot both temperature and rainfall. To plot data on multiple axis, some calculations are required since the plot has only one unique coordinate system. Plotting data in a new axis therefore requires to scale the additional data (and the corresponding axis) to the coordinate system of the plot.

Below is an example with the climate data from https://en.wikipedia.org/wiki/Fribourg:

```
> temp <- c(-0.9, 0.5, 3.9, 7.5, 11.6, 15, 17, 16.5, 13.7, 8.8, 3.8, 0.3)
> rain <- c(75, 70, 63, 64, 77, 95, 85, 111, 81, 63, 85, 75)
> par(las=1, mar=c(4,4,0.5,4))
> plot(1:12, temp, type='b', col='red', pch=19, xlab="Month", ylab="Average temperature
> scale <- 10
> lines(1:12, rain / scale, type='b', col='blue', pch=16)
> labels <- pretty(0:max(rain))
> axis(side=4, at=labels / scale, labels=labels)
> mtext("Average rainfall [mm]", side=4, line=3, las=3)
```

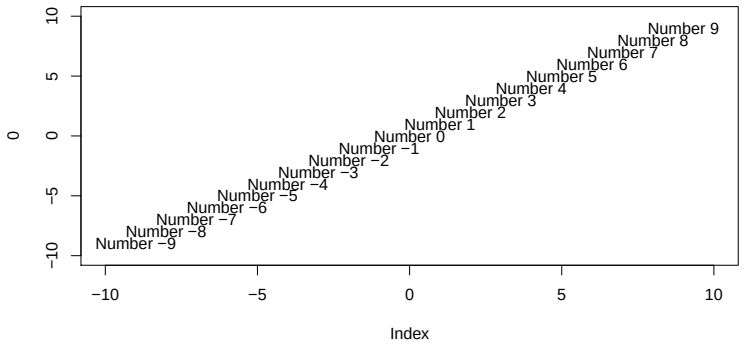


9.1.3 Adding Text to a Plot

In addition to `lines()` and `points()`, you may also add text to a plot using the function `text()` that takes the coordinates where text is to be put, followed by the argument `labels` that specifies the labels to be printed.

Before illustrating this, let us introduce another important concept: empty plots. You can generate an empty plot with `plot()` by setting `type='n'` (n for none). Creating empty plots is particularly helpful when building up plots using loops. But back to `text()`:

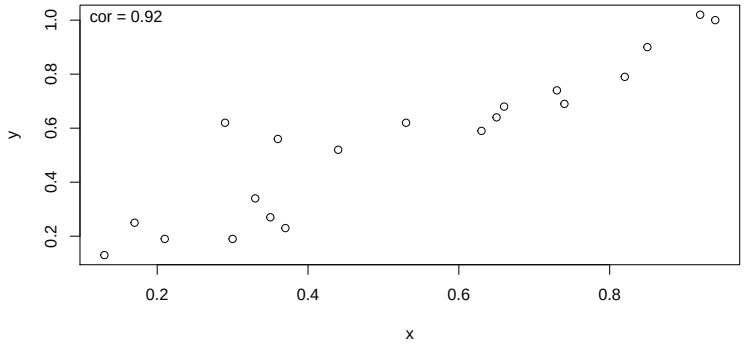
```
> plot(0, type='n', xlim=c(-10, 10), ylim=c(-10, 10))
> text(-9:9, -9:9, labels=paste("Number", -9:9))
```



We may also use the `text()` call to add extra legends to the plot. For instance, we might want to indicate the correlation between two data sets in the top left corner. Remember that we can use the `par("usr")` call to get the coordinates of the plot, allowing us to add text at fixed visual positions regardless of the `xlim` and `ylim` choices.

```
> x <- c(0.17, 0.63, 0.37, 0.94, 0.73, 0.74, 0.33, 0.36, 0.65, 0.82, 0.29, 0.13, 0.85, 0.21, 0.30
> y <- c(0.25, 0.59, 0.23, 1.00, 0.74, 0.69, 0.34, 0.56, 0.64, 0.79, 0.62, 0.13, 0.90, 0.19, 0.19
```

```
> plot(x, y)
> text(par("usr")[1], par("usr")[3] + 0.95 * (par("usr")[4]-par("usr")[3]), labels=past
```

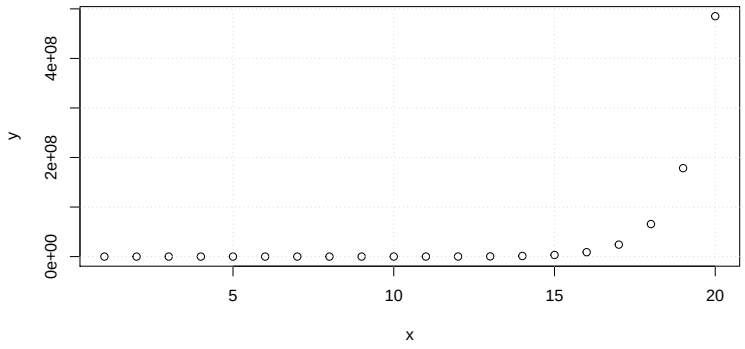


9.1.4 Reference lines

For readability, it is sometimes helpful to add reference lines to a plot. Here we consider two types of reference lines: grids and, well, reference lines.

Let's start with the grid. You can add a grid using the function `grid()` that will place fine dashed lines at the values indicated by tick marks. It takes all all sorts of arguments, including `lty` and `col`, and a way to choose the number of grid lines to draw.

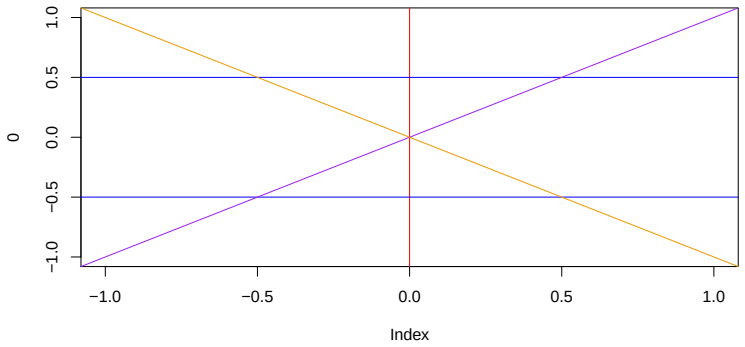
```
> x <- 1:20
> y <- exp(x)
> plot(x, y)
> grid()
```



In some cases, you may just want to add specific lines. This is easily done with `abline` that plots straingt lines to a plot.

- To plot horizontal lines, use the argument `h` to specify the y-values.
- To plot vertical lines, use the argument `v` to specify the x-values.
- To plot any other line, use either `a`and `b` denoting the intercept and slope, or `coef` to provide a vector of length two with said values.

```
> plot(0, type='n', xlim=c(-1,1), ylim=c(-1,1))
> abline(v=0, col='red')
> abline(h=c(-0.5,0.5), col='blue')
> abline(a=0, b=1, col='purple')
> abline(coef=c(0,-1), col='orange2')
```



9.1.5 Exercises: Plotting

See Section 18.0.21 for solutions.

1. Create a data frame with three columns x, y, and z as follows: x contains a vector with 1.0, 1.5, 2.0, 2.5, ... 100, each element of y is $y_i = \sqrt{(x_i)}$, and each element of z is $z_i = log(x_i)$. Plot y and z against x in the same plot, using a red solid line for y and a blue dashed line for z. Add the axis labels "x" and "A transformation of x" and a legend specifying the transformation.

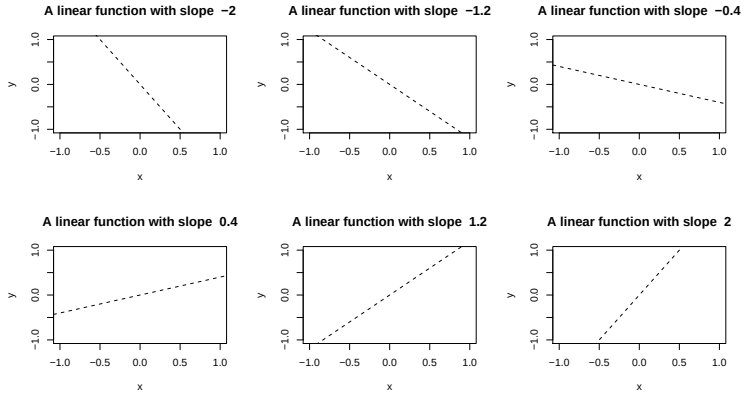
2. Open an empty plot with `xlim=c(-10,10)` and `ylim=c(-1,1)` and axis labels "x" and "y". Plot a vertical and a horizontal dashed red line crossing the oprigin (0,0). Put the Roman numbers I, II, III and IV to name the https://en.wikipedia.org/wiki/Quadrant_(plane_geometry).

3. Plot a climate diagram for Bagui, Central African replublic showing the average daily high temperature `av.high <- c(32, 34, 34, 32, 32, 30, 32, 30, 30, 30, 32, 32)` and the average percipitation `av.rain <- (16, 31, 104, 131, 161, 155, 193, 225, 192, 199, 76, 27)`. Put the two quantities on seperate axis and make sure that their maximum is at the same height in the plot. Add a legend.

9.2 Multiple panels

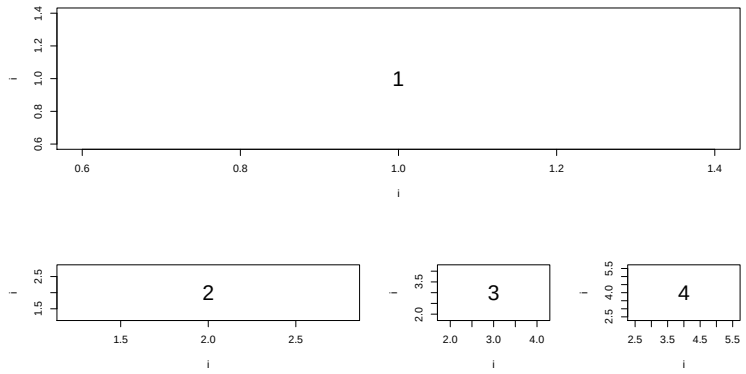
Often it may be desirable to create figures with multiple panels. That is rather straight forward with the `par()` option `mfrow` that specifies a grid of panels to be used via a vector with the desired number of rows and columns.

```
> x <- seq(-10, 10, length.out=31)
> par(mfrow=c(2,3), pch=16, lty=2)
> for(slope in seq(-2, 2, length.out=6)){
+    plot(0, type='n', main=paste("A linear function with slope ", slope), xlab="x", yl
+   abline(a=0, b=slope)
+ }
```



If you want to plot multiple figures in a more unique composition, the `layout()` function is very helpful. Just as the `mfrow` option to `par()`, `layout()` divides the plotting area up into as many rows and columns as desired. However, you can further specify the relative `heights` and `widths` of each column and have some plots using multiple cells of your grid. The first argument of `layout()` is a matrix with the grid and the panel indexes, best explained with an example:

```
> layout(matrix(c(1, 1, 1, 2, 3, 4), nrow=2, byrow=TRUE), width=c(2, 1, 1), heights=c(
> for(i in 1:4){
+    plot(i, i, pch=as.character(i), cex=2)
+ }
```



Once you set some options with `par()` or `layout()`, you may wonder how to unset them. There are two possibilities: you overwrite them with another call to either `par()` or `layout()`, or you close the graphics device altogether and start

anew. To close a graphics device, use `dev.off()`. You will learn about graphics devices in the next section.

9.2.1 Exercises: Plotting Multiple Pannels

See Section 18.0.22 for solutions.

1. Create a data frame with three columns x, y, and z as follows: x contains a vector with 1.0, 1.5, 2.0, 2.5, ... 100, each element of y is $y_i = \sqrt{(x_i)}$, and each element of z is $z_i = log(x_i)$. Plot y and z against x in two panels next to each other and use titles to identify the transformations.
2. Use a `for` loop and `par()` to create a plot with 12 pannels arranges in a 3x4 layout. Each panel should plot the log of the values `x <- 1:100` for different bases of `bases <- 2:13`. Each plot should use a different color and a different plotting symbol. Write the base used to the lower-right corner.
3. Use the layout function to plot four figures of x against a, one in the top row, two in the middle row and again one in the bottom row. Use any colors and symbols that you like.

Chapter 10

Probability Distributions

As a statistical language, it is no surprise that R comes with many tools for working with probability distributions, as well as implementations of most standard distributions. In this chapter, we will introduce these tools, focusing first on the most general of all distributions: the normal or Gaussian distribution.

10.0.1 The normal distribution

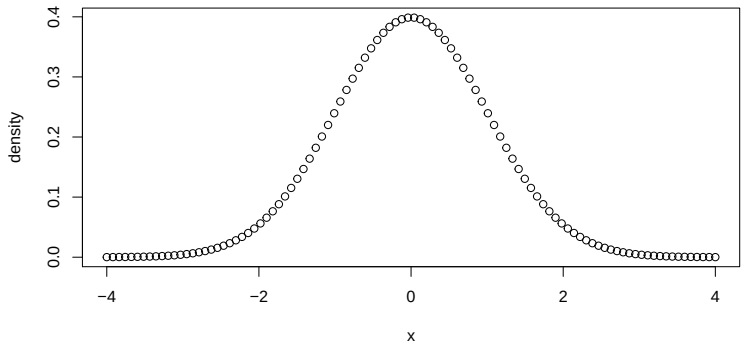
The normal distribution is defined through the density function

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ denotes the mean (first moment) and σ^2 the variance (second moment). If $\mu = 0$ and $\sigma^2 = 1$, the resulting distribution is termed the standard normal distribution, usually denoted by $\varphi(x)$.

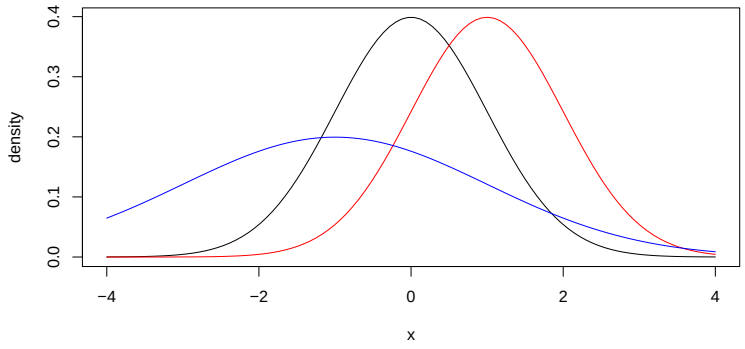
Let's first use R to have a look at this density function. For this, we will use the function `dnorm()` that returns the density of the standard normal distribution at any point.

```
> x <- seq(-4, 4, length.out=100)
> plot(x, dnorm(x), ylab="density")
```



The function `dnorm()` can also be provided the `mean`and `sd` of the normal distribution, that is, the mean μ and the standard deviation $\sigma = \sqrt{\sigma^2}$.

```
> plot(x, dnorm(x), ylab="density", type='l')
> lines(x, dnorm(x, mean=1), col='red')
> lines(x, dnorm(x, mean=-1, sd=2), col='blue')
```



In addition to the density function, R provides three equally important functions for the normal distribution:

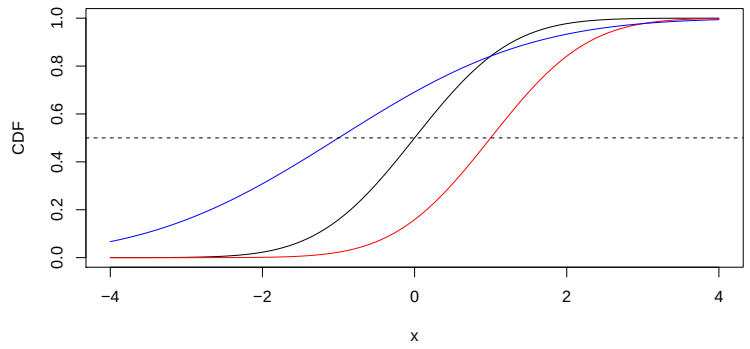
- The cumulative distribution function (CDF) $F(x|\mu,\sigma^2) = \int_{-\infty}^{x} f(x|\mu,\sigma^2)\mathrm{d}x$, which is the integral of the probability density function from minus infinity up to x. The corresponding R function is `pnorm()`.
- The quantile function $Q(q|\mu,\sigma^2) = F^{-1}(q|\mu,\sigma^2)$ that, given a quantile q, specifies the value of x such that $F(x) = q$. The corresponding R function is `qnorm()`.
- A function to generate random samples from the normal distribution: `rnorm()`.

The cumulative distribution function gives us important information about the location of a value within the distribution, also called **quantile**, with 0 implying the far left and 1 the far right. Specifically, it gives us $F(x|\mu,\sigma^2) = P(X \leq x|\mu,\sigma^2)$ which is the probability that a random value from the distribution X is

smaller or equal to x.

To illustrate this, let's plot the quantiles:

```
> plot(x, pnorm(x), ylab="CDF", type='l')
> lines(x, pnorm(x, mean=1), col='red')
> lines(x, pnorm(x, mean=-1, sd=2), col='blue')
> abline(h = 0.5, lty = 2)
```



We see that the cumulative distribution function is monotonically increasing within the interval [0,1]. This means that for infinitely small values of x, the CDF is zero: $\lim_{x\to-\infty} F(x) = 0$; and for infinitely large values of x, the CDF is one: $\lim_{x\to\infty} F(x) = 1$. All other values of x will hence be somewhere in between zero and one.

In addition, since a normal distribution is symmetric around the mean, $F(\mu|\mu,\sigma^2) = \frac{1}{2}$ in all cases. This is visible in the plot, where the dashed line crosses the curves exactly at their mean.

```
> pnorm(0)
[1] 0.5
> pnorm(1, mean=1)
[1] 0.5
> pnorm(-2, mean=-2, sd=10)
[1] 0.5
```

However, values far away from the mean have quantiles that are either close to zero or one.

```
> pnorm(1)
[1] 0.8413447
> pnorm(1, mean=5)
[1] 3.167124e-05
```

By definition, the quantile function `qnorm()` is the inverse of the cumulative function:

```
> x <- 2
> q <- pnorm(x)
> qnorm(q) == x
[1] TRUE
```

We can use `qnorm()` to answer the question: What is the value of x for the p^{th} quantile of the normal distribution? For example, what is the value of x for the 50th quantile of the standard normal distribution?

```
> qnorm(0.5)
[1] 0
```

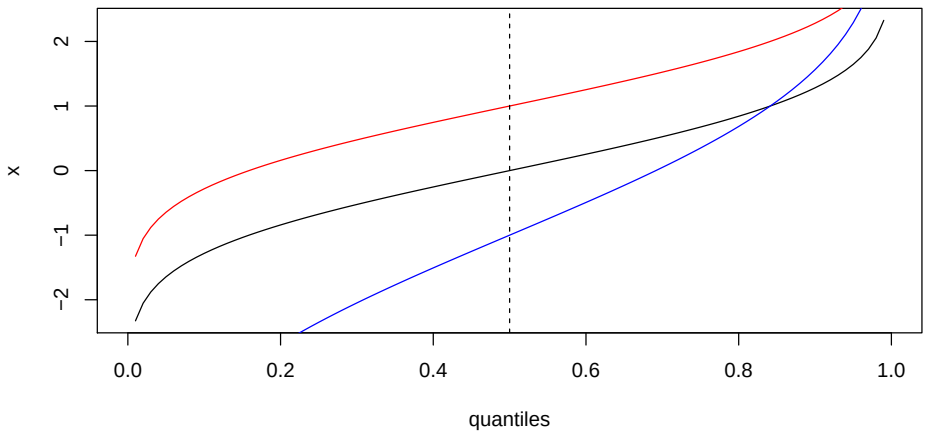
The answer is zero, since the mean (= the 50th quantile) of the standard normal distribution is zero. Or, what is the value of x for the 99th quantile of the standard normal distribution?

```
> qnorm(0.99)
[1] 2.326348
```

This results in a large value of x, which makes sense since 99% of all random values from the distribution are smaller or equal to x.

We can of course also plot the inverse cumulative distribution function:

```
> quantiles <- seq(0, 1, by = 0.01)
> plot(quantiles, qnorm(quantiles), ylab="x", type='l')
> lines(quantiles, qnorm(quantiles, mean=1), col='red')
> lines(quantiles, qnorm(quantiles, mean=-1, sd=2), col='blue')
> abline(v = 0.5, lty = 2)
```



Again, the dashed line at the 50%-quantile intersects the curves exactly at their mean. In fact, this is the same plot as above for `pnorm()`, except that the x- and y-axis are switched!

The last function `rnorm()` returns a vector of random numbers from a normal distribution. Its first argument `n` specifies the length of the random vector

returned and is mandatory.

```
> rnorm(1)
[1] 0.9189405
> rnorm(10, mean=100)
 [1]  99.98132  98.64246 100.54770 100.83919  98.24976 100.11746 100.11344
 [8]  98.65078 100.43653  97.71879
```

Note that since these numbers are drawn at random, your numbers will likely be somewhat different. We will talk more about the issue of randomness below.

10.0.2 Other Distributions

As mentioned above, R implements basically all standard probability distributions using the same syntax:

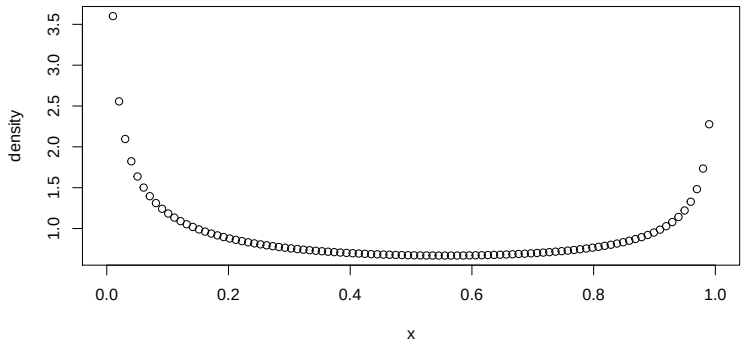
- The probability density function starts with a `d` followed by an (abbreviated) name referring to the distribution.
- The cumulative density function starts with a `p` followed by an (abbreviated) name referring to the distribution.
- The quantile function starts with a `q` followed by an (abbreviated) name referring to the distribution.
- The function to generate random numbers starts with a `r` followed by an (abbreviated) name referring to the distribution.

Among the many distributions implemented, the following are noteworthy:

- The uniform distribution with name `unif` with parameters `min`and `max`.
- The Binomial distribution with name `binom` with parameters `size` and `prob` that refer to the number of trials and the probability of a success.
- The exponential distribution with name `exp` and parameter `rate` (usually written as λ).
- The Poisson distribution with name `pois` with parameter `lambda`.
- The Beta distribution with name `beta` with parameters `shape1`and `shape2` (usually written as α and β).

To plot the beta distribution, for instance, we can use

```
> x <- seq(0, 1, length.out=100)
> plot(x, dbeta(x, 0.5, 0.6), ylab="density")
```



Note that all distribution functions take vector input. To generate random values from a Poisson distribution with different rates, you may provide a vector of lambdas:

```
> rpois(10, lambda=seq(1, by=20, length.out=10))
 [1]   0  19  43  79  87 100 120 147 168 181
```

Sometimes you may want to simulate outcomes according to some fixed probabilities. For instance, you may want to simulate the cases 1,2 and 3 with probabilities 0.1, 0.7 and 0.2, respectively. As we have seen before, this can be achieved with `sample()`

```
> x <- sample(1:3, size=1000, prob=c(0.1,0.7,0.2), replace=TRUE)
> sum(x==2) / length(x) # approximately 0.7
[1] 0.707
```

In case we only have two possible outcomes, we may also use the uniform distribution between 0 and 1. Indeed, the probability that a random value $X < p$ is just p. Hence, a vector of outcomes with probability $p = 0.73$ can be simulated as

```
> x <- runif(1000) < 0.73
> sum(x) / length(x) # approximately 0.73
[1] 0.729
```

10.0.3 Setting the Seed

Random numbers have the inherent property to be hardly reproducible. While this is often a desired property, it can be very cumbersome for debugging, since the results change every time we run a script.

To generate random numbers, `R` uses an algorithm that starts with an initial number (the so-called seed), and then generates consecutive numbers based on this. The seed hence completely defines the sequence of pseudo-random

numbers. By default, `R` uses the current time as a seed. Therefore, when we run e.g. `runif()` twice, it results in different numbers, since the time and therefore the seed is different:

```
> runif(5)
[1] 0.88136217 0.39878620 0.03879324 0.47187387 0.39366050
> runif(5)
[1] 0.4968277 0.2221195 0.5084117 0.3607840 0.8513912
```

However, we can define our own seed with the function `set.seed()`. It expects a number as an argument that will be used as a seed. What number we use doesn't matter at all. `R` creates the same random numbers every time we run the script:

```
> set.seed(13); runif(5)
[1] 0.71032245 0.24613730 0.38963444 0.09138367 0.96206454
> set.seed(13); runif(5)
[1] 0.71032245 0.24613730 0.38963444 0.09138367 0.96206454
```

Using a different seed will again create different random numbers:

```
> set.seed(14); runif(5)
[1] 0.2540337 0.6378273 0.9571886 0.5525467 0.9830671
```

10.0.4 Exercises: Probability distributions

See Section 18.0.23 for solutions.

1. What is the probability density of $x = 0.15$ under an exponential distribution with rate $\lambda = 2$?
2. What is the probability density of $x = 3$ under a Poisson distribution with $\lambda = 2$?
3. What is the value x such that a random value from a normal distribution with mean $\mu = 5$ and variance $\sigma^2 = 4$ is smaller than x with a 99% probability?
4. What is the probability density of $x = 0.72$ under a beta distribution with both shape parameters $\alpha = 0.7$ and $\beta = 0.7$?
5. What is the value x such that a random value from a standard normal distribution is larger than x with a 30% probability?
6. Generate 100 random values from a Poisson distribution with $\lambda = 2$.
7. Generate 13 random values from a exponential distribution with rate 0.5.
8. Simulate 10^3 values from a standard normal distribution. What is the fraction of those values that are < -1? What is the theoretical expectation? Do you get closer to the theoretical expectation with more random draws?

9. Simulate 10^3 values from a binomial distribution with size 10 and probability 0.1. What is the fraction of those values that are > 2? What is the theoretical expectation?

10. Human heights are approximately normally distributed, with a mean of 164.7 cm and a standard deviation of 7.1 cm for females and a mean of 178.4 cm and a standard deviation of 7.6 cm for males. Plot the two probability density functions (use heights ranging from 140cm to 200cm). Add a vertical line at your own height to the plot. What is the probability density of your height?

11. Let's revisit the normal distributions for human height from the previous exercise. Plot the cumulative distribution function for both male and female heights, using the same parameters as above. Add a line at your own height to the plot. What is the quantile for your own height (i.e. what is the probability to be smaller or equal your height)?

12. Let's revisit the normal distributions for human height from the previous exercise. What male or female heights represent the 5%, 32%, 50%, 68% and 95% quantile, respectively? Again plot the probability density functions, and add these heights as vertical lines in order to illustrate the quantiles.

13. Replicate the two plots regarding the probability density function and the cumulative distribution function on the Wikipedia page for the exponential distribution.

14. Simulate a random walk as follows:

    1. Initialize the random walk with $x_1 = 0$
    2. Sample the next value $x_{t+1} \sim \mathcal{N}(x_t, 1)$ from a normal distribution with mean x_t of the previous value and variance $= 1$.
    3. Repeat step 2 until you reach x_{100}.
    4. Plot the full random walk as purple filled dots connected by lines.

15. Simulate the Martingale System for betting. In this system, a player bets on a color in a roulette game and thus has a 18/37 chance of winning, in which case the invested money is doubled (if you invest 4 CHF and win, you'll have 8 CHF). The idea of the martingale system is to start with a modest investment of 1 CHF. Whenever you loose, you double the invest for the next round. As soon as you win, you stop the game. Simulate playing roulette according to this system until the player won or had to invest more than 100 CHF. Do such simulations 1000 times. How often did the player win? What was the average net benefit across replicates?

10.1 Plotting empirical distribution

Above we have seen how to plot theoretical probability distributions. But we might be equally interested in visualizing some empirical distributions. Here we

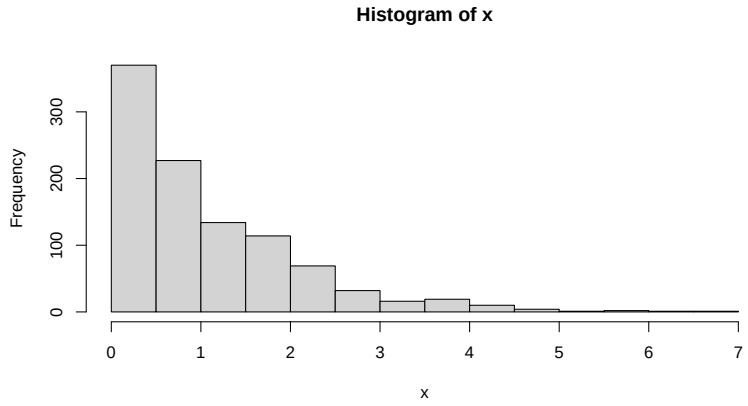
will introduce a few ways to do that.

10.1.1 Histograms

A common way to plot empirical distributions is via histograms. A histogram is basically a barplot in which each bar represents the fraction of values that fall within a particular bin. Building a histogram thus implies three steps: 1) deciding on the bins to use, 2) counting how many values fall within each bin, and 3) plotting these counts (or fractions) as a barplot.

The function `hist()` does all that.

```
> x <- rexp(1000)
> hist(x)
```



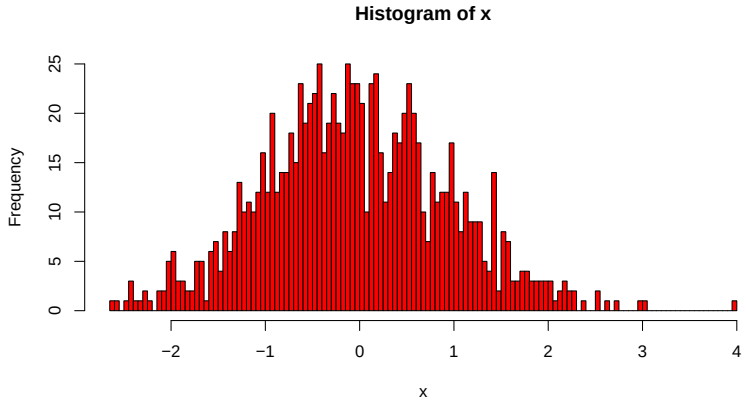
However, the optional arguments of `hist()` still allow for a fine control:

- `breaks`: specifies how bins are to be formed. It either takes a vector of break-points, the number of desired break-points, or some indication of how to compute break-points from the data. The default is a method introduced by H.A. Sturges in 1926 - still the best around.

- `freq`: when `TRUE`, frequencies (i.e. counts) are plotted, when `FALSE`, densities (i.e. fractions) are used.

- And pretty much all graphical parameters such as `col` work too.

```
> x <- rnorm(1000)
> hist(x, breaks=100, col='red')
```



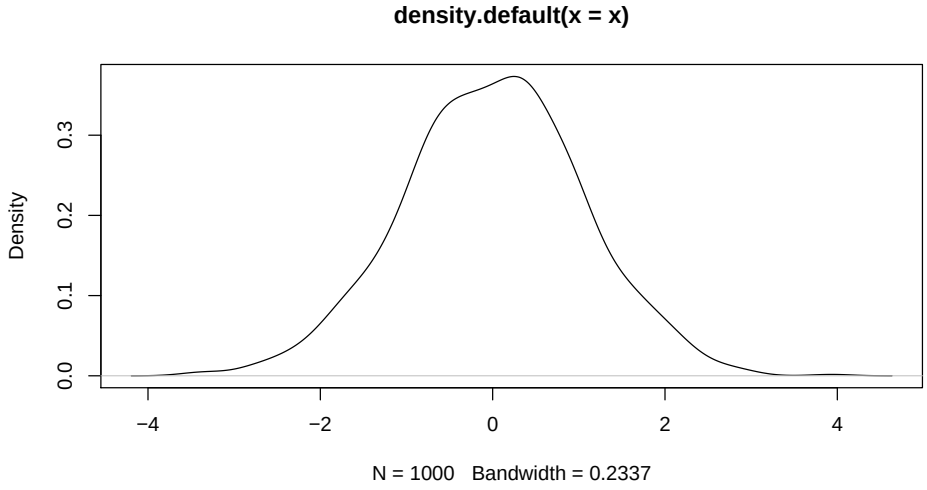
10.1.2 Kernel Density Estimation

Estimating probability densities from continuous numbers always involves the challenge that for any given value there are usually zero observations. Thus, densities can only reasonably be computed from values within an interval. The histogram introduced above is a way to do that: it partitions the space into a number of non-overlapping bins and calculates the density of each by counting the number of values that fall within each bin. By doing so it makes the assumption that all values within a bin are equally important to estimate its density, but any point outside of that interval is meaningless, no matter how close it is to the bin boundary.

Yes, that seems draconian: two points can be arbitrarily close to each other, but if they are on opposite sides of the boundary, one is given full weight and the other none.

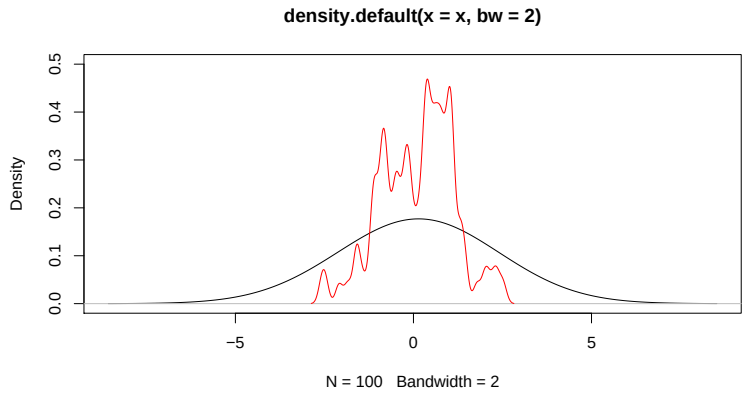
However, there are also other ways to estimate densities that use other metrics. A popular alternative (that is by no means less arbitrary) is kernel density estimation that uses some kernel function to give weight to values based on how far away they are of the location at which you want to estimate the density. In R, the function `density()` performs this task.

```
> x <- rnorm(1000)
> plot(density(x))
```



The key parameter is the bandwidth (argument `bw`), which indicates how broad the distribution giving weight should be. A wide distribution (i.e. a large bandwith) gives some weight to points even far away from the location at which the density is estimated. A narrow distribution (i.e. a small bandwidth) gives meaningful weights only to values rather close to that location. The choice of the bandwidth therefore affects the smoothing a lot.

```
> x <- rnorm(100)
> plot(density(x, bw=2), ylim=c(0, 0.5))
> lines(density(x, bw=0.1), col='red')
```

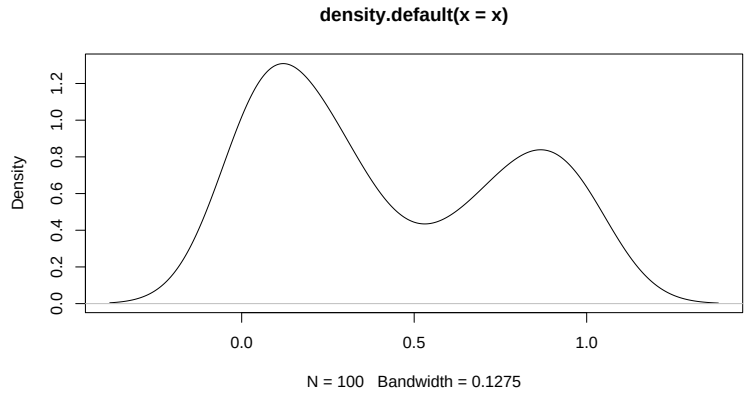


Luckily, R implements a clever algorithm to decide on an optimal bandwith and it is usually not advised to tinker too much with its choice. Rather, one might simply increase or decrease the proposed bandwith slightly using the argument `adjust`.

Just as histograms, kernel density estimation also has its drawbacks: it is particularly inaccurate at boundaries. As an example, consider samples from a

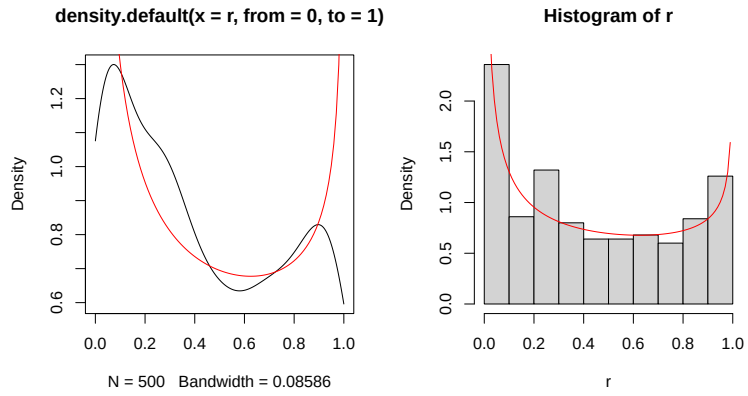
beta distribution, which is only defined between 0 and 1.

```
> x <- rbeta(100, shape1=0.5, shape2=0.7)
> plot(density(x))
```



If the limits are known, they may be provided to `density()` using the arguments `from` and `to`. However, in such cases the histogram is usually a better way to represent the data, as illustrated in the following where the density estimates are compared against the theoretical density function given as a solid red line.

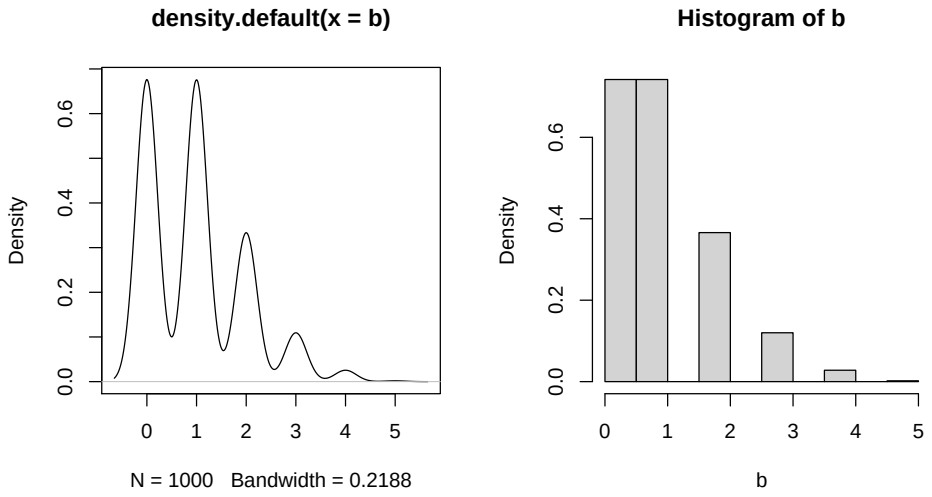
```
> r <- rbeta(500, shape1=0.5, shape2=0.7)
> par(mfrow=c(1,2))
> plot(density(r, from=0, to=1))
> x <- seq(0, 1, length.out=100)
> lines(x, dbeta(x, shape1=0.5, shape2=0.7), col='red')
> hist(r, freq=FALSE)
> lines(x, dbeta(x, shape1=0.5, shape2=0.7), col='red')
```



Another example are discrete distributions.

```
> b <- rpois(1000, lambda=1)
> par(mfrow=c(1,2))
```

```
> plot(density(b))
> hist(b, freq=FALSE)
```



In conclusion: both histograms and density plots are fair ways to visualize empirical distributions, but inferring probability densities is generally a dirty business.

10.1.3 Exercises: Empirical Distributions

See Section 18.0.24 for solutions.

1. Simulate 10^3 values from a binomial distribution with sample size 5 and probability of success $p = 0.2$. Visualize the empirical distribution using both a histogram and kernel density smoothing. Which one would you use?

2. Plot the theoretical probability density function of a normal distribution with mean 10 and standard deviation 2 as a solid red line. Randomly sample 10^1, 10^2, 10^3, 10^4 and 10^5 values from this normal distribution, and add the kernel density estimates on the same plot in black using different line types. Add a legend.

3. Load the built-in iris data set with `data("iris")`. R will store the data set into the variable `iris`, and you can inspect the data e.g. with `head(iris)`. Plot a separate histogram of petal lengths for each of the three species setosa, versicolor and virginica. Next, create a new plot containing the kernel density estimation for all three species together as a solid black line. Add lines for the kernel density estimation for each species separately to your plot, using different colors. Add a legend.

Chapter 11

Import & Export

It is important to be able to load data to R and to save any output you might need. R offers easy ways to do so, which we will introduce in this chapter.

But before doing so, we will need to talk briefly about the concept of **working directories**. Just as when working on the command line, R has a working directory in which, well, you work! Specifically, the working directory is the default place in your file system at which R looks for files you want to load and where it stores files you write.

You can find out your current working directory with the function `getwd()`. Note that your working directory might be different from mine.

```
> getwd()
[1] "/Users/xenia/git/runifr"
```

You can also easily change your working directory by using the function `setwd()` and providing either the full or relative path. Note again that different filesystems differ in how they are organized, and hence they offer different options.

On a Unix system (e.g. Linux and OS X), this should work

```
> getwd() # were I am now
[1] "/Users/xenia/git/runifr"
> a <- getwd() # storing where I am now
> setwd("~/") # changing to home folder
> getwd()
[1] "/Users/xenia"
> setwd(a) # changing back to original
> getwd()
[1] "/Users/xenia/git/runifr"
```

If you are on a Windows computer, relative paths will work the same, but your

full path will start with the drive letter, i.e. "C:" or "D:".

Rather than through the console, you can also change the working directory via RStudio under **Session** -> **Set Working Directory**, where you can either choose directly by browsing or switch to the directory of the current R script.

11.0.1 Exercises: Data Input and Export

See Section 18.0.25 for solutions.

1. What is your current working directory?
2. Create a directory of your choice somewhere, then switch your working directory there. Did it work?

11.1 Exporting and Importing Data Frames

11.1.1 Exporting

The most generic way to save a data frame is with the function `write.table()`.

```
> x <- data.frame(ID=1:3, Random=runif(3))
> write.table(x, file="random.txt")
```

This file will look like this:

```
"ID" "Random"
"1" 1 0.0742941116914153
"2" 2 0.961419784929603
"3" 3 0.447386628715321
```

Note that by default, `write.table()` adds the row index as row names (the numbers in quotes at the beginning). This behavior can be modified using the argument `row.names` that either accepts a vector with alternative row names, or can be set to `FALSE` to omit row names altogether. This is usually the preferred option.

```
> write.table(x, file="random.txt", row.names=FALSE)
```

The resulting file will now look like this:

```
"ID" "Random"
1 0.0742941116914153
2 0.961419784929603
3 0.447386628715321
```

Important other optional arguments are:

- `col.names`: works just like `row.names` but for the columns.
- `append`: when set to `TRUE`, will append the data to an exisiting file rather than overwriting it.

- `sep`: specifies the field separator to be used (default is a white space).
- `quote`: specifies whether characters and factors should be written in quotes (default is `TRUE`).

Here is a more complicated example:

```
> d <- data.frame(Number=c(1,3,5), Animal=c("Lion", "Chimpanzee", "Flamingo"))
> write.table(d, "animals.txt", sep = "\t", row.names=FALSE)
> # append to file
> f <- data.frame(a=c(7,2), b=c("Lynx", "Falcon"))
> write.table(f, "animals.txt", sep = "\t", row.names=FALSE, append=TRUE, col.names=FALSE)
```

The resulting file will look like this:

```
"Number"    "Animal"
1   "Lion"
3   "Chimpanzee"
5   "Flamingo"
7   "Lynx"
2   "Falcon"
```

11.1.2 Importing

A file such as this one can be read back from the file system using the function `read.table()`.

```
> read.table("animals.txt")
      V1         V2
1 Number     Animal
2      1       Lion
3      3 Chimpanzee
4      5   Flamingo
5      7       Lynx
6      2     Falcon
```

Note that by default, `read.table()` does not expect column names and will thus interpret the first row as data. To have the first row properly interpreted as column names, use the argument `header` and set it to `TRUE`.

```
> read.table("animals.txt", header=TRUE)
  Number     Animal
1      1       Lion
2      3 Chimpanzee
3      5   Flamingo
4      7       Lynx
5      2     Falcon
```

The function `read.table()` also offer many useful optional arguments:

- `sep`: specifies the field separator to be used (default is a white space).

- `nrows`: limit import to the first n rows only.
- `skip`: indicates the number of rows to skip before reading. This is helpful for files that have comments at the beginning.
- `stringsAsFactors`: whether or not R should convert strings into factors. The default value for `stringsAsFactors` depends on your R version (new versions have default `FALSE`). If `stringsAsFactors` is `TRUE`, the column Animal is turned into factors:

```
> read.table("animals.txt", header=TRUE, stringsAsFactors=TRUE)$Animal
[1] Lion       Chimpanzee Flamingo   Lynx       Falcon
Levels: Chimpanzee Falcon Flamingo Lion Lynx
```

If this is undesired, we may turn this conversion off:

```
> read.table("animals.txt", header=TRUE, stringsAsFactors=FALSE)$Animal
[1] "Lion"       "Chimpanzee" "Flamingo"   "Lynx"       "Falcon"
```

11.1.3 Other input and export options

While `write.table()` and `read.table()` are generic functions with many options that can handle all export and import problems, many options have to be tuned in some cases. This is why R offers two sets of functions for specific use cases:

- Comma separated values files, so-called CSV files, are a common data exchange format that can be imported and exported by Calc, Excel or other spreadsheet programs. In such files, each line contains one data record with values (columns) separated by a delimiter. In contrast to what the name suggests, most spreadsheet programs actually use a semicolon `;` and not a comma `,` as delimiter. While CSV files can be written and read using the standard function `write.table()` and `read.table()`, the functions `write.csv()`and `read.csv()` or `write2.csv()`and `read.csv2()` are easier to use as they offer better defaults. The difference between these is that the `.csv()` functions use commas `,` and the `.csv2()`functions use semicolons `;` as delimiters.

- Instead of commas or semicolons, many people also use tabs to separate data fields. For such files, the functions `write.delim()` and `read.delim()` offer more useful defaults.

Finally, R also offers the possibility to write files line-by-line using the function `cat()`. By default, `cat()` prints to the console.

```
> cat("Hello world!")
Hello world!
```

However, `cat()` has the optional arguments `file` and `append` that can be used to build files.

```
> cat("Hello world!", file="myfile.txt", sep="\n")
> cat(1:10, file="myfile.txt", append=TRUE)
```

The above code will create a file "myfile.txt" that looks like this

```
Hello world!
1 2 3 4 5 6 7 8 9 10
```

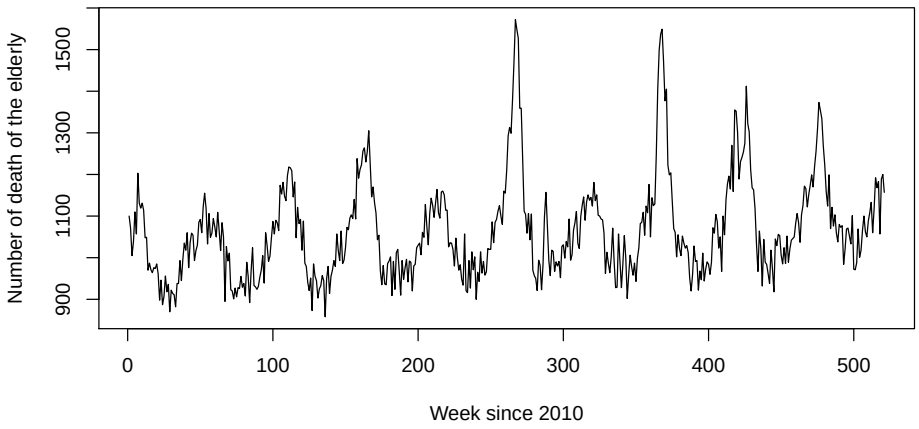
11.1.4 Importing from a URL

The functions introduced above can also read data provided via a URL. For this, the URL has to be turned into a connection using the function `url()`, and this connection can be provided to any reading function.

As an example, consider the weekly number of deaths happening in Switzerland as provided by Swiss Federal Statistical Office here.

The relevant CSV file can be directly imported into R.

```
> d <- read.csv2(url("https://www.bfs.admin.ch/bfsstatic/dam/assets/12607335/master"), stringsAsF
> plot(d$NumberOfDeaths[d$Age=="65+"], type='l', ylab="Number of death of the elderly", xlab="Wee
```



11.1.5 Exercises: Exporting and Importing

See Section 18.0.26 for solutions.

1. Create a data frame with the columns x, y, and z, each containing 100 standard normal random numbers. Write this data frame to your working directory as "my_data_frame.txt". Quit RStudio and use a text editor to add two additional entries (lines). Start RStudio again and import the modified data frame as d.

2. Open Calc (or Excel) and create a dummy file with some data. Find a way to import it into R using read.table(). (Hint: use the CSV format).

3. Load the population data on Swiss cities from (https://www.bfs.admin.ch/bfsstatic/dam/assets/12647632/appendix) into R and plot the population size of 1930 against that of 2018 for each city. Exclude the total population of Switzerland (first row).

11.2 Listing Existing Files

R also offers the possibility to list all files in a directory. This is particularly helpful when writing scripts that should parse over all files in a directory. Existing files are listed with the function `list.files()` that outputs a character vector containing a list of files in your working director, or in any other directory provided by the optional argument `path`.

```
> file_list <- list.files()
```

Often you may not want a list of all files but just those that match a particular pattern. This is easily achieved using the argument `pattern`.

```
> file_list_txts <- list.files(pattern="*.txt")
```

11.2.1 Exercises: Listing files

See Section 18.0.27 for solutions.

1. Calculate percentage of how many files in your Downloads directory are .pdf.

11.3 Exporting Graphics

RStudio makes it easy to export graphics: simply click on the "Export" icon in the "Plots" panel.

However, you can get much finer control over graphics export. When R produces graphical output, this output does not go to the console. Instead, it goes to what is known as a graphical **device**. That device might be the default R device shown in the "Plots" panel, but also a device that produces a graphics file such as a pdf, a svg, a png or a jpeg.

To choose one of these graphics devices, you need to first open it, then produce your plot, and then close it. To open one of the above listed graphics devices, you can use `pdf()`, `svg()`, `png()` or `jpeg()`. To close it, use `dev.off()`.

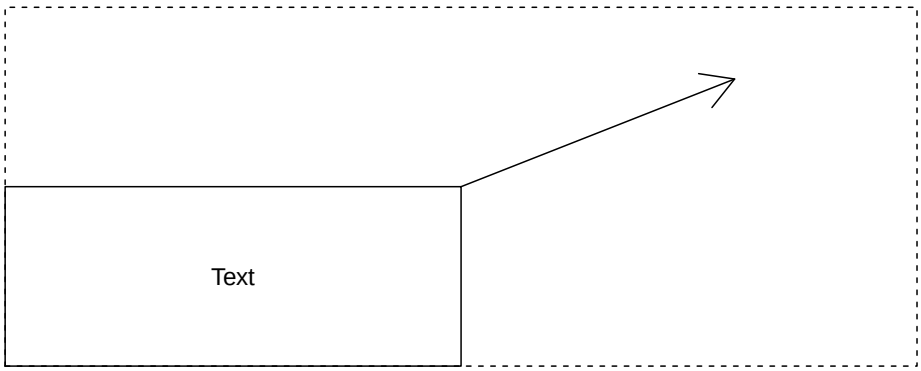
```
> pdf("test.pdf")
> par(mar=c(4,4,0.1,0.1))
> plot(1:10, col='dodgerblue', pch=5)
> dev.off()
pdf
  2
```

The above code will produce a pdf called "text.pdf" in your working directory. Note that to have an effect, any call to `par()` must run after the graphic device was opened.

All functions to write graphics files takes as their first argument the `filename`, which may also contain a path, of course. Important additional arguments are `width` and `height` that specify the dimensions of the resulting plot. For vector graphics (e.g. pdf and svg), `width` and `height` indicate the size of the plot in inches. For raster graphics (e.g. png and jpeg), they specify the dimension in pixels.

Changing the dimension of a plot also affects the size of plotted elements relative to the plot. The best way to understand what is happening is to think of the plot dimensions as the size of the paper used for plotting, but to not affect the size of the font, for instance. Here is an illustration of what is happening to text when the paper size is increased: it will remain at the same size, and hence be smaller relative to the size of the plot.

```
> plot(0, type='n', xlim=c(0,1), ylim=c(0,1), xaxt='n', yaxt='n', xlab="", ylab="", bty='n')
> rect(0, 0, 0.5, 0.5)
> text(0.25, 0.25, "Text")
> rect(0, 0, 1, 1, lty=2)
> arrows(0.5, 0.5, 0.8, 0.8)
```



To increase the quality of a jpeg or png plot, you can therefore not simply increase the number of pixels with `width` and `height`: you will need also fiddle with `pointsize` and scale it accordingly.

```
> png("low_resolution.png", width=200, height=200, pointsize=12)
> plot(1:10)
> dev.off()
pdf
  2
> png("high_resolution.png", width=2000, height=2000, pointsize=120)
> plot(1:10)
> dev.off()
```

```
pdf
  2
```

Things are a bit easier with vectorized graphics that do not have a resolution. Hence, you can use `width` and `height` to adjust the relative size of the plot without worrying about `pointsize`.

Note that we generally recommend vector graphics for quality.

11.3.1 Exercises: Exporting graphics

See Section 18.0.28 for solutions.

1. Sample a vector $\boldsymbol{x}$ of length 100 from a standard normal distribution $x_i \sim \mathcal{N}(0,1)$. Generate a corresponding vector $\boldsymbol{y}$ under the linear model $y_i = 4 + 1.5x_i + \epsilon_i$ where the error rates $\epsilon_i \sim \mathcal{N}(0, 0.2)$ are drawn randomly from a normal distribution with variance 0.2. Plot $\boldsymbol{y}$ against $\boldsymbol{x}$ in a pdf. Use `par()` to adjust the margins to have minimal white space.
2. Plot the same data again to another pdf, but change the dimensions to 2x2 inches. Appreciate how this makes everything much larger relative to the plot.
3. Plot the same data to two jpeg images with resolutions 480x480 and 1280x1280, but identical apart from the resolution.

11.4 RStudio Workspace

In the tab 'Session', you find options to Save and Load Workspace. These options allow you to save all variables (and their content) that you have created during your session.

11.4.1 Exercises: RStudio Workspace

See Section 18.0.29 for solutions.

1. Assign the values 1,2,3 to variables `a`, `b` and `c`, respectively. Quit RStudio and save the work space.
2. Start RStudio again and check that all variables you created are still there.
3. Use R to calculate `log(a)` and assign the result to variable `y`. Quit RStudio without saving the work space. Restart RStudio and check that the variables `a`, `b` and `c` exist, but not `y`.

Chapter 12

Packages

While base R comes loaded with heaps of useful functions, what really distinguishes R comes through its easy extendability. R extensions come as **packages** that can easily be installed (and uninstalled) and loaded if needed. And since pretty much everyone can contribute packages, there are tons of those out there.

So where to start? The place to go to is the Comprehensive R Archive Network (CRAN) available at https://cran.r-project.org/. However, as with most other things regarding R, finding help in the official documents is often cumbersome - so if you are looking for something specific, an internet search engine is usually your best friend.

Once you have identified a package you would like to install, you can use the function `install.packages()` to do so. For illustration, consider the package RColorBrewer that allows you to create nice color gradients that you can use for plotting.

```
> install.packages("RColorBrewer")
```

Once a package is installed, you will need to load it to make use of its functions. You do so with the function `library()`.

```
> library(RColorBrewer)
```

Note that the function `install.packages()` requires the name of a package in quotes (i.e. as a string), while `library()` requires it without. Don't ask me why!

Note also that base R already comes with many packages pre-installed. So you not always need to install a package - you might already have it.

12.0.1 Exercises: Installing packages

See Section 18.0.30 for solutions.

1. Install the package RColorBrewer on your system. Type `display.brewer.all()` into your console. You see all the color gradients RColorBrewer offers. Pick one you like. Generate a color palette with `brewer.pal` (check the help page on how to use this function) such that it contains 10 different colors.
2. Generate a vector `x` with values -4, -3.9, ..., 3.9, 4. Plot the normal density of `x` with mean zero and 10 different standard deviations 0.1, 0.2, ..., 0.9, 1, as lines inside one plot. Color each line with its own color from the gradient generated above!

Chapter 13

Writing Functions

Throughout this tutorial, you have been using tons of functions built into R, like `seq()`, `mean()`, `plot()`, `rnorm()` etc. However, you can easily write your own functions that perform specific tasks!

For example, say you've received a cool data set and want to run a bunch of statistical analyses. It's highly recommended to normalize variables before calculating certain statistics. Normalization will squeeze all values to be between 0 and 1, and hence enables a fair comparison of data with different magnitudes.. The formula for normalization is:

$$\bar{x} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

This formula can easily be implemented in R like this:

```
> x <- rnorm(5)
> (x - min(x)) / (max(x) - min(x))
[1] 1.000000000 0.003232373 0.578723932 0.000000000 0.304387379
```

Now, imagine you continue writing code and encounter a second vector `y`, which also needs to be normalized. Of course, you could copy-paste your code from above:

```
> y <- rnorm(5)
> (y - min(y)) / (max(x) - min(y))
[1] 0.0000000 0.4487176 0.6120174 0.4286857 0.3490270
```

However, this is very bad practice. For example, you might forget to change a variable while copy-pasting. Did you spot the mistake in the formula above? We forgot to replace `max(x)` by `max(y)`!

The alternative to copy-pasting is writing custom functions. This has three main advantages:

1) You can give a function a name that makes it easy to understand what it does. For example, guessing the purpose of a function called `normalize()` would be pretty straightforward.

2) If you have a bug in your code, or want to add extra steps, you only need to change your code in one place. Imagine you had implemented the function for normalization wrongly in the first place - e.g. it should be $x_{min} - x_{max}$ in the denominator - and then copy-pasted it 10 times. You would have to fix this bug 10 times in our code!

3) By writing code only once inside a function, you eliminate copy-paste mistakes.

Convinced? Let's have a look on how to implement our normalization function!

```
> normalize <- function(x){
+   frac <-  (x - min(x)) / (max(x) - min(x))
+   return(frac)
+ }
```

There are 4 steps to consider when writing own functions:

1) Name: You need pick a name for your function. This can be any valid object name, ideally something that speaks for itself. As functions always DO something, it is recommended that you pick a verb for your function name - for example, `normalize()` if the function normalizes variables, `permute()`, `calculateSummaryStats()` etc. Although you could use names of existing functions (e.g. create a custom `min()` function), you should avoid this as it causes ambiguities.

2) Arguments: You list the inputs (also called arguments) to the function inside the brackets of `function()`. The input for the example above is simply `x`. You can specify as many inputs as you want, e.g. `function()` if there are no arguments, or `function(x, y, z)`, if there are three.

3) Actions: What should the function do with the inputs? Calculate a statistic, as above? Plot something? You write all the real R code inside the body of the function, i.e. inside the curly brackets `{}`. In the above example, we calculate the variable `frac`, which contains the normalized values of `x`.

4) Output: What should the function return when it's finished with the actions? We can return vectors, matrices, data frames, lists etc. Just place the object to return inside `return()` - for example, we return a vector `frac` above. A function can also return nothing (by just omitting `return()`), for example if you want to plot something. Note that we can only return a single R object, writing something like `return(x, y, z)` does not work.

In summary:

```
> NAME <- function(ARGUMENTS) {
+
```

```
+   ACTIONS
+
+   return(OUTPUT)
+
+ }
```

Let's go back to our example `normalize()`. Try to execute the code where we define the function. Nothing should happen, but R now knows about the function, such that it is ready to be used! We can call the function as every other function we've used before:

```
> normalize(x)
[1] 1.000000000 0.003232373 0.578723932 0.000000000 0.304387379
```

. . . and if we ever encounter other vectors that should be normalized, we can simply call the function again:

```
> normalize(y)
[1] 0.0000000 0.7331779 1.0000000 0.7004468 0.5702893
```

Another advantage of functions is that if our requirements change, we only need to make the change in one place. For example, you might encounter variables including `NaN` and infinite values. In these cases, `normalize()` fails:

```
> x <- c(x, NaN, Inf)
> normalize(x)
[1] NaN NaN NaN NaN NaN NaN NaN
```

Because we've extracted the code into a function, we only need to make the fix in one place:

```
> normalize <- function(x){
+   min_x <- min(x, na.rm = TRUE, finite = TRUE)
+   max_x <- max(x, na.rm = TRUE, finite = TRUE)
+   frac <-  (x - min_x) / (max_x - min_x)
+   return(frac)
+ }
>
> normalize(x)
[1]   0   0   0   0   0 NaN NaN
```

This is an important part of the "do not repeat yourself" (or DRY) principle. The more repetition you have in your code, the more places you need to remember to update when things change (and they always do!), and the more likely you are to create bugs over time.

13.0.1 Exercises: Writing Functions

See Section 18.0.31 for solutions.

1) Write a function that returns the square of a variable
2) Write a function that converts Fahrenheit to Celsius.
3) Write functions to compute the sample variance and skewness of a numeric vector. Sample variance is defined as

$$\text{Var}(x) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

where $\bar{x} = \frac{1}{n}\sum x_i$ is the sample mean. Skewedness is defined as:

$$\text{Skew}(x) = \frac{\frac{1}{n-2}\sum_{i=1}^{n}(x_i - \bar{x})^3}{\text{Var}(x)^{3/2}}$$

4) Write `calculateBothNA()`, a function that takes two vectors of the same length and returns the number of positions that have an `NA` in both vectors.

13.1 Input to functions

13.1.1 Default arguments

When you create functions with many inputs, you'll probably want to start adding default values. Arguments with default values do not need to be specified by the user, as R will simply take the default. Most functions that you've used so far have default values. For example, type `?hist` into the console. You can see that only the first argument `x` does not have a default value, while all other arguments, such as `breaks`, `main`, `xlab` etc. do have default values. Including defaults can save the user a lot of time because it keeps him from having to specify every possible input to a function.

Specifing default values in the function definition is very easy, just add `=` and then the default value after the input argument. For example, we could write a function that calculates the density of the square of a normal distribution. We want the arguments `mean`, `sd` and `log` to have default values:

```
> calculateDensitySquareOfNormal <- function(x, mean = 0, sd = 1, log = FALSE){
+   return(dnorm(x, mean, sd, log)^2)
+ }
```

Now, R will set any inputs that the user does not specify to its default:

```
> x <- 0.6
> calculateDensitySquareOfNormal(x)
[1] 0.1110386
> calculateDensitySquareOfNormal(x, 1.473)
[1] 0.07427313
> calculateDensitySquareOfNormal(x, 1.473, 2.1)
```

```
[1] 0.0303618
> calculateDensitySquareOfNormal(x, 1.473, 2.1, TRUE)
[1] 3.053005
```

Note that if we wanted e.g. to change `sd` to 2.1, but leave `mean` at its default value, we can not simply write

```
> calculateDensitySquareOfNormal(x, 2.1)
[1] 0.01677481
```

as R would interpret 2.1 as the input for mean. We either need to specify both `mean` and `sd`, or then (better!) name the arguments:

```
> calculateDensitySquareOfNormal(x, 0, 2.1)
[1] 0.03326051
> calculateDensitySquareOfNormal(x, sd = 2.1)
[1] 0.03326051
```

13.1.2 Dot-dot-dot (...) - the wildcard argument

Many functions in R take an arbitrary number of inputs:

```
> sum(1, 2, 3)
[1] 6
> sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
[1] 55
```

How do these functions work? They rely on a special argument: `...` that captures any number of arguments that aren't otherwise matched.

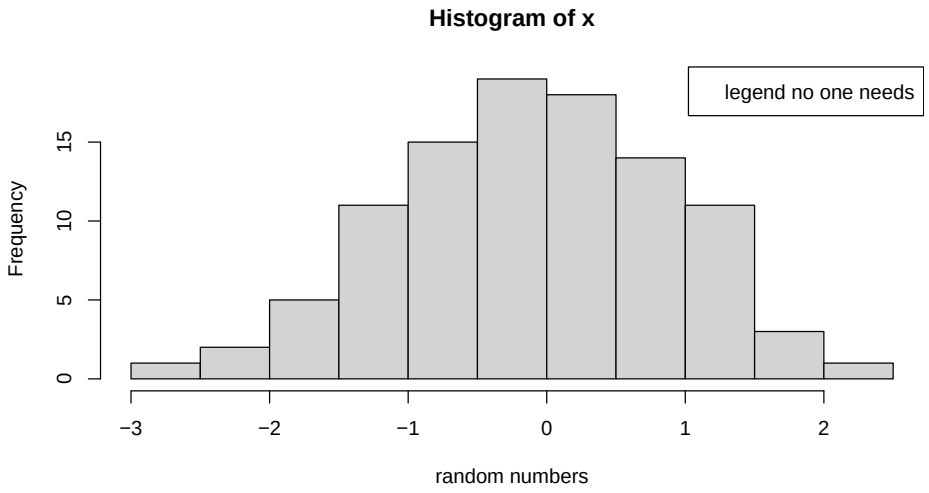
This becomes useful when you send those `...` on to another function, e.g. when your function primarily wraps another function. For example, if you create a custom function that calls the histogram function `hist()` in R, you might also want the user to be able to specify optional inputs for the plot, like `main`, `xlab`, `ylab`, etc. However, it would be a real pain to have to include all possible plotting parameters as inputs to our new function:

```
> plotHistWithLegend <- function(x, text, breaks = "Sturges",
+       freq = NULL, probability = !freq,
+       include.lowest = TRUE, right = TRUE,
+       density = NULL, angle = 45, col = NULL, border = NULL){
+
+   # call hist()
+   hist(x, breaks = breaks, freq = freq, probability = probability, include.lowest = include.low
+   # do extra stuff
+   legend("topright", legend = text)
+ }
```

... and these are not even all parameters `hist()` accepts! Thankfully, we can

take care of all of this by using `...` as an input to the function. Note that the `...` notation will only pass arguments on to functions that are specifically written to allow for optional inputs. Luckily, `hist()` allows this (check the help page). The `...` input tells R that the user might add additional inputs that should be used later in the function. Hence, `...` is a very useful catch-all for function wrappers. Let's re-write our function accordingly:

```
> plotHistWithLegend <- function(x, text, ...){
+   # call hist()
+   hist(x, ...)
+   # do extra stuff
+   legend("topright", legend = text)
+ }
> plotHistWithLegend(rnorm(100), "legend no one needs",
+                    xlab = "random numbers")
```



Whatever parameters we give to our function `plotHistWithLegend()` (for example `xlab`), it will just pass them on to `hist()`, which will use them if it finds a matching argument.

13.1.3 Exercises: Input to functions

See Section 18.0.32 for solutions.

1) Write a function that returns a variable raised to the power of another variable: x^y. Variable `y` should have a default argument.

13.2 Checking values

By default, once a function is called, all code inside the function will be executed. However, there may be cases where there's no point in evaluating parts of the

code and it's best to stop everything and leave the function altogether. This might not seem so important at the moment as we've only dealt with a small number of neat little functions. But imagine you're writing complex code, maybe an entire R package, that will be used by people that don't know logic. How will you make sure that these users call every function the way you want it to be called?

As an example, let's create a new function `calculateWeightedMean()` that calculates the mean of vector `x` using vector `weights` as weights:

```
> calculateWeightedMean <- function(x, weights){
+     return(sum(x * weights) / sum(weights))
+ }
```

What happens if `x` and `weights` are not the same length?

```
> calculateWeightedMean(1:9, 1:3)
[1] 16
```

Although the functions runs and produces a result (as R recycles the shorter vector `weights`), this might not be what you want. Instead, you would like to tell the user he's doing something wrong, and then stop the execution of the code, instead of wasting time for a useless result. We can achieve this with the function `stop()`. If R ever executes a `stop()` function, it will directly quit the function it's currently evaluating, and print an error message. You can define the exact error message as an argument for `stop()`:

```
> calculateWeightedMean <- function(x, weights) {
+   if (length(x) != length(weights)) {
+     stop("Wow, stop!! Arguments 'x' and 'weights' must have the same length!")
+   }
+   return(sum(x * weights) / sum(weights))
+ }
```

Let's call the function with an invalid argument - i.e. 2 vectors of different length:

```
> calculateWeightedMean(1:5, 100:10000)
Error in calculateWeightedMean(1:5, 100:10000): Wow, stop!! Arguments 'x' and 'weights' must have
```

With valid arguments, the function will run as before:

```
> calculateWeightedMean(1:5, 5:9)
[1] 3.285714
```

13.2.1 Exercises: Checking values

See Section 18.0.33 for solutions.

1) Write a function that converts Kelvin to Fahrenheit. Consider that the absolute zero temperature is 0 Kelvin, hence negative values are invalid.

Throw an error if a user enters invalid values!

13.3 Return values

13.3.1 Implicit return

As mentioned above, we can return values from a function using `return()`. However, if you omit the `return()` statement, R will still return something - simply the last statement it evaluates:

```
> normalize <- function(x){
+   (x - min(x)) / (max(x) - min(x))
+ }
>
> x <- rnorm(5)
> normalize(x)
[1] 0.0000000 0.2443474 1.0000000 0.1218673 0.7942871
```

Although it is good to know that this feature exists, it is recommended to always explicitely use `return()`. Return is good tool for clearly designing "leaves" of code where the routine should end, jump out of the function and a return value.

13.3.2 Multiple return statements

Inside a function, you will often encounter conditionals (`if`/`else` statements) that handle specific cases. Depending on the case, you might want to return something different - i.e. use multiple `return()` statements:

```
> normalize <- function(x) {
+   if (length(x) == 0)
+     return(0)
+   else
+     return((x - min(x)) / (max(x) - min(x)))
+ }
```

13.3.3 Returning multiple values

As we have seen above, `return()` only returns a single R object. If you want to return multiple values from a function, you need an R object that stores multiple values. A very convenient object in `list()`, as it can store objects of different classes. In addition, you can name these objects, which comes in handy when evaluating the return values of a function. Say we want a function that calculates summary statistics on data, but all in once:

```
> calculateSummaryStats <- function(x){
+   mean <- mean(x)
+   variance <- var(x)
```

```
+   stddev <- sd(x)
+   allInOne <- list(mean = mean, variance = variance, stddev = stddev)
+   return (allInOne)
+ }
```

We calculate mean, variance and standard deviation of our data, bind the results to a list and return the list. We can now extract the results easily:

```
> result <- calculateSummaryStats(rnorm(100))
>
> result$mean
[1] 0.1028083
> result$variance
[1] 1.210136
> result$stddev
[1] 1.100062
```

13.3.4 Exercises: Return Values

See Section 18.0.34 for solutions.

1) Program the factorial function $n! = n * (n-1) * (n-2) * ... * 1$ with a for loop.

2) Program the factorial function using the function `prod()`.

3) Write a function to simulate rolling a dice 100 times and to count the number of 6's (use a loop). Return both the simulated numbers and the total number of 6's.

4) Again write a function to simulate rolling a dice 100 times and to count the number of 6's. But now, use as few instructions as possible. Can you try to avoid using a loop?

5) Implement a fizzbuzz function. It takes a single number as input. If the number is divisible by three, it returns "fizz". If it's divisible by five it returns "buzz". If it's divisible by three and five, it returns "fizzbuzz". Otherwise, it returns the number. Make sure you first write working code before you create the function.

13.4 Environment

When you start writing more complicated code, you might get confused at some point about the scope of variables. First, you should know that variables that are defined within a function are only available within this function, but not from outside. For example, consider this function:

```
> computeSquare <- function(x) {
+   square <- x^2
+   return(square)
+ }
```

The variable `square` is local to our function `computeSquare()` and can not be accessed from outside:

```
> computeSquare(3)
[1] 9
> print(square)
Error in print(square): object 'square' not found
```

On the other hand, we can use variables inside a function that are not necessarily defined within the function. For example, take this function:

```
> computeSum <- function(x) {
+   return(x + y)
+ }
```

In many programming languages, this would be an error, because `y` is not defined inside the function. In R (and python, too), this is valid code. R will first search for the value of `y` inside the function (the smallest scope). If it can't find it there, R will look for `y` in the environment.

```
> y <- 100
> computeSum(10)
[1] 110
```

This behaviour is sometimes a bit confusing, so it's good to be aware of it.

13.4.1 Exercises: Environment

See Section 18.0.35 for solutions.

1) Consider the following code:

```
> x <- 20
> computeSum <- function(x, y = 10) {
+   z <- x + y
+   return(z)
+ }
```

Given the above code was run, which value does `computeSum(x = 4, 6)` produce?

Chapter 14

Thinking in Vectors

R is interpreted at runtime, making it relatively slow compared to compiled or semi-compiled languages such as C++ or Java. To compensate for that, most functions are implemented in a faster language (often C++) and R just calls these functions to do the job. If this feature is exploited well, R might turn out to be actually quite fast.

In this section we will discuss how to make R code run fast by thinking in vectors (or matrices) rather than loops. If you are familiar with other programming languages this may feel strange at first. And if you are new to programming know at least this: it is an R thing.

14.0.1 Loops are slow

To measure the benefit of thinking in vectors, let us first introduce a way to measure execution time. This is done using the function `system.time()`.

```
> system.time(runif(10^6))
   user  system elapsed
  0.011   0.002   0.014
```

The function `system.time()` returns three measures of time:

1. The total user time the CPU spent on the task.
2. The total system time the CPU spent on the task.
3. The total time that elapsed from start to end.

Usually, the elapsed time is much larger than the CPU time as the CPU is also used by other processes on the system or may need to wait for some data to be loaded. The difference between the user and system time is a bit arbitrary and OS specific, but that is not of concern here.

What is of concern, is that generating 10^6 random numbers using a `for` loop

will be much slower than the `runif(10^6)` call above, regardless of the measure used:

```
> getRand <- function(n){
+   x <- numeric(n);
+   for(i in 1:n){
+     x[i] <- runif(1)
+   }
+ }
> system.time(getRand(10^6))
   user  system elapsed
  1.610   0.283   1.901
```

Why is that? It is not due to the random number generation **per se** as both calls generate 10^6 of those. Instead, the difference is due to R intrinsics: the `for` loop makes 10^6 calls to the function `runif()`, while in the first case there is a single call. And in many cases, function calls are much slower than the actual mathematical problem to solve.

Here is another example.

```
> x <- runif(10^6);
> system.time(x*x)
   user  system elapsed
  0.001   0.000   0.001
> system.time(for(i in 1:10^6){x[i]*x[i]})
   user  system elapsed
  0.026   0.001   0.027
```

You might be tempted to think that the loop version also needs to increment the variable `i`. However, any implementation of the piece-wise product `x*x` will also need to do that. The difference is therefore only whether this happends in R or "inside" the function.

What that means for us as programmers: avoid loops in R whenever possible. Instead, try to execute commands on whole vectors (or matrices).

Since many R functions take vectors as arguments, this possible for many cases by just being a little creative. Here is an example to generate a 10x1000 matrix of 10 data sets (rows) that are all drawn from a normal distribution but with means 0, 1, ..., 9 and standard deviations 1, 2, ..., 10.

```
> X <- matrix(rnorm(10*1000, mean=rep(0:9, 1000), sd=rep(1:10, 1000)), nrow=10)
> mean(X[2,])
[1] 1.044023
> sd(X[5,])
[1] 4.858478
```

The trick here is to provide the matching mean and sd for each element of the matrix in one call, rather than using a loop. So, let's be creative!

14.0.2 Exercises: Loops are slow

See Section 18.0.36 for solutions.

1. A particularly expensive function is `exp()`. Generate a vector $\boldsymbol{x}$ of 10^6 uniform random numbers $x_i \sim \mathcal{U}(0,1)$ to see how much slower `exp(x)` is compared to `x+x` when these functions were called on the entire vector.
2. Write a function that takes a vector and returns its sum $\sum_i x_i$ using a `for` loop. Time your function using the same vector $\boldsymbol{x}$ from above and compare it to using `sum(x)`.
3. Write a function similar to that above to calculate $\sum_i \exp(x_i)$ using a `for` loop. Is it much slower than the function returning just the sum from the question above?
4. Generate a 100x10 matrix with 100 data sets (columns) each drawn from a Poisson distribution but with different $\lambda = 1, 2, \ldots, 10$ (one per column).

14.1 Apply

As we have seen above, loops are relatively slow in R. Luckily, many functions accept vectors (and matrices or data.frames) as input and do their magic on each element individually, which allows us to avoid writing loops.

However, not all functions do that. In addition, we may no always want to apply a function to each element, but maybe to each column or row of a matrix or data.frame.

Luckily, R comes with a family of functions that help us avoiding loops altogether. We will discuss the most important ones here.

We will ilustrate some of these concepts using a simple simulation task. Imagine we would like to simulate a simple dice game in which it matters how many sixes the player obtains when rolling 5 dice. What is the distribution of 0, 1, ... or 5 sixes?

Obviously, this is a binomial sampling problem. Knowing this, we can readily obtain these probabilities using `dbinom()`:

```
> dbinom(0:5, size=5, prob=1/6)
[1] 0.4018775720 0.4018775720 0.1607510288 0.0321502058 0.0032150206
[6] 0.0001286008
```

But for the sake of the argument, let's try to get those using simulations. A `for` loop implementation could look like this (note that R indexes start at 1):

```
> simSixes <- function(dice, rep){
+   counts <- integer(6)
+   for(i in 1:rep){
+     roll <- sample(1:6, size=dice, replace=TRUE)
```

```
+     numSixes <- sum(roll==6)
+     counts[numSixes+1] <- counts[numSixes+1] + 1
+   }
+   return(counts / rep)
+ }
> system.time(sim <- simSixes(dice=5, rep=10^5))
   user  system elapsed
  0.436   0.013   0.450
> sim
[1] 0.39957 0.40358 0.16092 0.03269 0.00315 0.00009
```

How can we speed up these simulations?

14.1.1 `replicate()`

The most basic function to avoid a `for` loop is `replicate()`. As the name suggests, it replicates a particular function call. It takes the following main arguments:

- The number of replicates `n`.
- The expression `expr` to be called.

We could use `replicate()` to generate many random numbers as follows:

```
> replicate(10, runif(1))
 [1] 0.6929201 0.4358046 0.7050972 0.3200239 0.8760117 0.6720674 0.3047773
 [8] 0.1398616 0.5652658 0.6980336
```

That is obviously not super helpful as the function `runif()` has a built-in mechanism to simulate an arbitrary number of random numbers. But we can illustrate the benefit of using `replicate()` over a `for` loop for our case of rolling dice:

```
> simRoll <- function(dice){
+   roll <- sample(1:6, size=dice, replace=TRUE)
+   return(sum(roll==6))
+ }
> simSixesReplicate <- function(dice, rep){
+   rolls <- replicate(rep, simRoll());
+   counts <- tabulate(rolls, nbins=dice+1)
+   return(counts / rep)
+ }
> system.time(sim <- simSixesReplicate(dice=5, rep=10^5))
   user  system elapsed
  0.537   0.021   0.560
> sim
[1] 0.40376 0.20082 0.05372 0.00827 0.00072 0.00004
```

Since `replicate()` returns a vector of results, we need to tabulate those into counts using `tabulate()`.

Note that in this case, using `replicate()` did not result in a major speed-up. That is common for many tasks: while `replicate()` is a convenient function to write code cleaner, it does usually do little in terms of speed.

14.1.2 `apply()`

An often more useful function is `apply()` which applies a specific function to a data.frame or matrix. It takes the following arguments:

- The data structure `X` to which a function should be applied.
- The `MARGIN` over which to loop. `MARGIN=1` corresponds to rows, `MARGIN=2` to columns.
- The function `FUN` to be applied.
- Additional parameters `...` forwarded to `FUN`.

What `apply()` does is basically to loop over `X` along the `MARGIN` (i.e. along rows or columns) and applying `FUN` to each vector of that margin (i.e. to each row or column). Here is an example:

```
> m <- matrix(runif(1000), nrow=10)
> apply(m, 1, sum)
 [1] 50.98664 47.96165 46.53472 49.42322 46.78828 55.02994 46.73263 50.38601
 [9] 49.15007 47.85920
```

In the example above, we created a matrix with ten rows and used `apply()` to apply the function `sum()` to each row. This call to `apply()` does returns a vector of length 10, one value per row.

Now let's see how we could make use of `apply()` to make our dice simulations faster.

```
> simSixes <- function(dice, rep){
+   rolls <- matrix(sample(1:6, size=dice*rep, replace=TRUE), ncol=dice)
+   numSix <- apply(rolls == 6, 1, sum)
+   counts <- tabulate(numSix, nbins=dice+1)
+   return(counts / rep)
+ }
> system.time(sim <- simSixes(dice=5, rep=10^5))
   user  system elapsed
  0.161   0.006   0.169
> sim
[1] 0.40449 0.15945 0.03282 0.00283 0.00008 0.00000
```

As you can see, we now managed to significantly reduced computation time by making use of `apply()`. But let's walk through the code to understand how it works.

The basic idea is to avoid multiple function calls at all levels, so we should aim for generating all random draws at once. In total, we roll `dice*rep` times a dice, which we can do fast with `sample(1:6, size=dice*rep, replace=TRUE)`. However, we still need to preserve that we always consider `dice` rolls together. One way to do that is to store the resulting rolls in a matrix with `dice` columns such that each row corresponds to a roll of `dice` dice. Once we have this structure, we can use `apply()` to perform some calculations on a per row basis. Here, we want to use `sum()` to count the number of sixes, for which we first turn the integer matrix into a logical matrix using `rolls == 6` and then `apply()` to calculate the number of `TRUE`'s in each row.

14.1.3 `lapply()` and `sapply()`

R offers a bunch of similar functions to `apply()` that differ in what input they accept and which output the produce. The `apply()` function, for instance, does not deal with lists, but the `lapply()` functions does precicely that:

```
> l1 <- as.list(1:5)
> l2 <- lapply(l1, runif)
> l2
[[1]]
[1] 0.845548

[[2]]
[1] 0.5016538 0.5085640

[[3]]
[1] 0.03651228 0.36896596 0.17814951

[[4]]
[1] 0.7455067 0.1054734 0.3717697 0.8702687

[[5]]
[1] 0.1525040 0.7820551 0.9155104 0.2130255 0.1950921
```

In the example above, we used `lapply()` to generate a list of vectors of random numbers of different sizes, which could not be stored in a matrix.

Note that while `lapply()` always returns a list, it may not require a list as input.

```
> lapply(1:3, runif)
[[1]]
[1] 0.6604787

[[2]]
[1] 0.9738182 0.9781264
```

```
[[3]]
[1] 0.8911678 0.5364177 0.8744750
```

Sometimes, we may want to apply a function to a list but not obtain the results in list form but as a vector or matrix. In these cases we may use `sapply()`:

```
> sapply(l2, sum)
[1] 0.8455480 1.0102178 0.5836277 2.0930185 2.2581871
```

14.1.4 `sweep()`

The last such function we will discuss here is `sweep()` which sweeps out a summary statistics from a higher dimensional structure. The classic example is if you want to centralize some data to have a mean of zero.

Consider a matrix $\boldsymbol{X}$ of 100 samples (rows) obtained from 10 replicates (columns). The goal is now to standardize each replicate (column) to have mean zero. We can get the means of these data sets using `apply()`.

```
> X <- matrix(runif(1000), ncol=10)
> means <- apply(X, 2, mean)
```

But how can we now centralize each column? Well, that is trivial with a `for` loop:

```
> for(i in 1:ncol(X)){
+   X[,i] <- X[,i] - means[i]
+ }
```

We can avoid that `for` loop by expanding the means to match the dimensionality of $\boldsymbol{X}$:

```
> M <- matrix(rep(means, nrow(X)), ncol=ncol(X), byrow=TRUE)
> X <- X - M
```

A more elegant way is offered by `sweep` that takes the following arguments

- The data structure `x` to which a function should be applied.
- The `MARGIN` over which to loop.
- The vector of statistics `STATS` to be used.
- The function `FUN` to be applied.

To centralize our data, we can thus use `sweep()` as follows.

```
> X <- sweep(X, MARGIN=2, STATS=means, FUN="-")
```

In this call, we ask `sweep()` to sweep across columns (`MARGIN=2`) and to use the statistics stored in `means` using the index of the column, and to apply the operator `-`. Note that since `-` is an operator not a function name, we need to put in in quotes as `FUN="-"`.

14.1.5 Shortcuts

Some `apply()` calls are so frequently used, that R provides shortcuts to them. These include `rowSums()`, `colSums()`, `rowMeans()` and `colMeans()` that correspond to `apply()` calls with the appropriate `MARGIN` and `FUN` set to `sum` or `mean`.

```
> X <- matrix(runif(20), ncol=5)
> rowSums(X)
[1] 2.072556 2.692741 1.432204 1.928534
> apply(X, 1, sum)
[1] 2.072556 2.692741 1.432204 1.928534
> colMeans(X)
[1] 0.2220654 0.4735840 0.2865934 0.6079728 0.4412933
> apply(X, 2, mean)
[1] 0.2220654 0.4735840 0.2865934 0.6079728 0.4412933
```

14.1.6 Exercises: Apply

See Section 18.0.37 for solutions.

1. Use `lapply()` to create a list with 100 vectors as follows: sample for each vector an integer n_i from 1, 2, ..., 1000. The vector should then contain the value n_i 10 times.
2. Create a 11x1000 matrix $\boldsymbol{X}$ with rows of 1000 random numbers each, drawn from a normal distribution with means `-5:5` and standard deviations `seq(0.1, 2, length.out=11)`, one per row. Verify that the rows have the proper means and standard deviations using `apply()`.
3. Standardize the matrix $\boldsymbol{X}$ from above such that each row has mean zero and standard deviation 1. Verify with `apply()`.
4. Write a function `sumBelow()` that takes as input a vector $\boldsymbol{x}$ and a constant a. It then returns the sum of all elements of $\boldsymbol{x}$ that are $<$ than a. Use `apply()` to apply our functions to each data set of the standardized matrix $\boldsymbol{X}$ from the exercise above using $a = 0$ and $a = 1$.
5. Use `apply()` to generate a plot with four panels showing histograms of the data sets 4, 5, 6 and 7 of $\boldsymbol{X}$

Chapter 15

S3 Classes

So far, we've only done functional programming with R. This means we've created variables and functions and manipulated them. However, we can do object oriented programming in R, too! Object-oriented programming (OOP) is a computer programming practice in which programmers can define custom data structures. Such a data structure usually contains some data to store (so called *attributes*), as well as *methods*. We call this data structure a **class**. We can think of a class like a blueprint of something, say a student. A class "student" has attributes like a name, an age, a field of study, some grades, and so on. In addition, a class has methods that act on these attributes. Methods can be seen as "skills" of a class. For example, a student is able to say his name, age and field of study, or to compute his mean grade.

A class is a blueprint for an **object**. From our class "student" we can hence create an actual student object that would have a specific name, say Liam. Actually, we can create as many student objects as we want from the blueprint class "student". We could therefore create more student objects from our class, for example Susan, Paul, Carl etc. The class is the abstract blueprint, the object is the actual instance of a class.

While most programming languages have a single class system, R has three class systems. Namely, S3, S4 and more recently Reference class systems. In this tutorial, we'll focus on S3 classes, as this is the most popular and prevalent class.

15.1 The class attribute

In fact, everything in R is treated as an object. An object often has some attributes associated with it. These attributes can be seen as labeled values you can attach to an object. As an example, we can create a factor and inspect his attributes with the command `attributes()`:

```
> fac <- factor(c("1", "2", "3"))
> attributes(fac)
$levels
[1] "1" "2" "3"

$class
[1] "factor"
```

We can see that a factor object has two attributes: levels and class. The attribute `class` now tells you from which "blueprint" (class) this object was built. You may realize now that all factors we've used so far were in fact objects of the class "factor", a built-in class of R!

The attribute `class` is where S3 objects come in. In fact, an S3 object is simply an object that has a `class` attribute (and possibly other attributes to store data). You can see the class attribute specifically using the `class()` command:

```
> class(fac)
[1] "factor"
```

In fact, we can even change the class of an object over the `class` attribute!

```
> class(fac) <- "somethingElse"
> class(fac)
[1] "somethingElse"
```

15.2 Generic functions

We've previously discussed how to define attributes, especially the `class` attribute, of an object. We will now discuss how to define methods of a class. As an example, we're going to investigate the function `print()`.

`print()` is not a normal function; it is a **generic** function. This means that the function is written in a way that lets it do different things in different cases. You've already seen this behavior in action (although you may not have realized it). `print()` does one thing for numeric vectors:

```
> x <- c(1,2,3,4)
> print(x)
[1] 1 2 3 4
```

However, if we print a data frame, the output looks different:

```
> df <- data.frame(x = x)
> print(df)
  x
1 1
2 2
```

```
3 3
4 4
```

and so does the output of a factor:

```
> fac <- factor(x)
> print(fac)
[1] 1 2 3 4
Levels: 1 2 3 4
```

How can `print()` do this? You may imagine that `print()` looks up the `class` attribute of its input and then uses multiple `if` statements to pick which output to display. `print()` does something very similar, but much more simple. When you call `print()`, R examines the class of the input that you provide for `print()`. Then, R passes all arguments to another function that is specifically designed to handle that class of input.

For example, when you give `print()` a data.frame object, R will call another function, `print.data.frame()`. This function is specifically designed to print data frames. However, when you give `print()` a factor object, R will call another function, `print.factor()`. This function now knows how to print a factor object. Hence, `print.data.frame()` and `print.factor()` work like regular R functions, just like the ones we've seen before. There's nothing magic in them. However, each was written specifically so `print()` could call it to handle a specific class of print input.

In conclusion, the output of `print()` looks different for different class objects. Let's now have a closer look on the mechanisms that are behind this process.

A generic function usually has a very simple code. For example, the code for the generic function `print()` looks like this:‘

```
> print <- function (x, ...) {
+   UseMethod("print", x)
+ }
```

When you call `print()`, this calls a special function, `UseMethod()`. `UseMethod()` examines the class of the input that you provide for `print()`, and calls another function designed to handle that class of input. For example, when you give `print()` a data.frame object, `UseMethod()` will call `print.data.frame()`.

How does `UseMethod()` find the functions that can handle a specific class of input? Or, in other words, how does `UseMethod()` know that `print.data.frame()` is the function that it should call when printing a data frame?

The answer is that functions that handle a specific class of input (let's call them S3 methods) have a special name. In fact, every S3 method has a two-part name. The first part of the name will refer to the function that the method works with (e.g. “print”). The second part will refer to the class (e.g. “data.frame”). These two parts will be separated by a period (`.`). So for example, the `print()` method

that works with data frames is called `print.data.frame()`. The `print()` method that works with factors is called `print.factor()`. And so on.

When `UseMethod()` is called, it searches for an R function with the correct S3-style name. The function does not have to be special in any way; it just needs to have the correct name.

15.3 Writing S3 classes

Let's now create our own S3 class! We will go back to the example from the introduction and create a class "student". A student will have attributes name, age and field of study, as well as a vector of grades. A student can introduce himself and compute his average grade.

We start by creating an object that holds these attributes of a student. Then, we assign it to a class "student".

```
> liam <- list(name = "Liam", age = 25, field = "biology", grades = c(4.5, 6, 5.5))
> class(liam) <- "student"
```

Now let's write an S3 print method for the student class. This class needs to be named `print.student()`; otherwise `UseMethod()` will not find it. The method should also take the same arguments as `print()`; otherwise, R will give an error when it tries to pass the arguments to `print.student()`:

```
> print.student <- function(x, ...) {
+   output <- paste0("Hello there! I'm a student. My name is ", x$name, ", I'm ", x$age
+   print(output)
+ }
```

Does our method work? Yes, and not only that; R uses the print method to display the contents of `liam`.

```
> print(liam)
[1] "Hello there! I'm a student. My name is Liam, I'm 25 years old and I study biology
```

Note that we don't need to write the generic function `print()` itself. This method has already been implemented in base R.

Let's now implement a generic function `calculateMeanGrade()`. There is no generic R function that does this already, so we have to implement 1) the generic function `calculateMeanGrade()` and 2) the specialized function `calculateMeanGrade.student()`!

```
> # implement generic function
> calculateMeanGrade <- function(stud) {
+   UseMethod("calculateMeanGrade", stud)
+ }
>
```

```
> # implement S3 function
> calculateMeanGrade.student <- function(stud){
+   meanGrade <- mean(stud$grades)
+   return(meanGrade)
+ }
>
> # calculate mean grade
> calculateMeanGrade(liam)
[1] 5.333333
```

Note that you should never directly call the function `calculateMeanGrade.student()`. Although this would give you exactly the same result, the whole point of S3 classes and generic functions would be lost. Maybe you have other classes - e.g. "secondaryStudent", "universityStudent" - at some point that also need to calculate mean grades, but require a different implementation. The beauty of S3 programming is that you don't need to remember the class of the object you're dealing with. No matter if it is a secondaryStudent, a universityStudent or simply a student - you can just call `calculateMeanGrade()` and R will do whatever it needs to do for you.

15.4 Multiple classes

R objects can have multiple classes. Let's say our student `liam` is not only a student, but also a human! We can simply append the extra class to the `class` attribute:

```
> class(liam) <- c("student", "human")
```

A human will have a print function, too. As all humans usually have names and an age (but not a field of study and grades, as students do), the print function will look different:

```
> print.human <- function(x, ...) {
+   output <- paste0("Hey! I'm a human called ", x$name, " and I'm ", x$age, " years old.")
+   print(output)
+ }
```

What will happen when we print `liam`? Which function will be called - the print function for students or the print function for humans? The answer is that R will simply go one by one through the elements in the vector of `class` attributes. It will first look for a print function in the class `student`, as this is the first class in the vector. If it does find a print function for students, R will call it and return. If for some reason, the class `student` does not have a print function, R will go to the second element in the vector. It will hence search for a print function for humans. If it does find one, R will call this one instead. Therefore, if we call `print()` on `liam`:

```
> print(liam)
[1] "Hello there! I'm a student. My name is Liam, I'm 25 years old and I study biology
```

R still outputs the `print.student()` function, as student is the first class in the objects class vector. Now, let's reverse the order of the class vector, such that `human` comes before `student`. You will see that R now calls the `print.human()` function:

```
> class(liam) <- rev(class(liam))
> print(liam)
[1] "Hey! I'm a human called Liam and I'm 25 years old."
```

If neither student nor human had a `print()` method, R would call `print.default()`, a special method written to handle general cases. We can quickly demonstrate this by removing the two print methods we've defined above from the environment. This can be done with the command `rm()`:

```
> rm(print.student)
> rm(print.human)
```

R has forgotten about these functions. If we now call `print()` on our object, R will call the default print method:

```
> print(liam)
$name
[1] "Liam"

$age
[1] 25

$field
[1] "biology"

$grades
[1] 4.5 6.0 5.5

attr(,"class")
[1] "human"   "student"
```

15.5 Constructors

It is good practice to use a function with the same name as class to create objects. This function is called the *constructor* of a class, because it constructs an object of this class. Actually, you've already used a lot of constructors during this tutorial. For example, when you type

```
> df <- data.frame(x = x)
```

then `data.frame()` is the constructor - it takes some input and returns an object of class data.frame!

Constructors are nice because they make code easily readable. In addition, you can add some integrity checks on the member attributes. Let's go back to the student example and create a constructor for this class:

```
> student <- function(name, age, field, grades) {
+   # check if function arguments are valid
+   if (age < 0)
+     stop("Age of student must be > 0!")
+   if (any(grades < 0 | grades > 6))
+     stop("Grades must be between 0 and 6!")
+
+   s <- list(name = name, age = age, field = field, grades = grades)
+
+   # now set the class attribute
+   class(s) <- "student"
+   return(s)
+ }
```

We can now construct as many students as we want, in a very simple and clean way!

```
> # all ok
> liam <- student("Liam", 26, "biology", c(4.5, 6, 5.5))
> marco <- student("Marco", 28, "physics", c(5, 4, 5.5))
>
> # not ok!
> carl <- student("Carl", -25, "philosophy", c(5.5, 5, 4))
Error in student("Carl", -25, "philosophy", c(5.5, 5, 4)): Age of student must be > 0!
```

In summary, to make a class (best practice):

1) Write a constructor for your class.
2) Write class methods for any generic function likely to use objects of your class.

The primary use of object-oriented programming in R is for print, summary and plot methods. These methods allow us to have one generic function, e.g. `print()`, that displays the object differently depending on its type. Imagine that you as an R user would have to call `print.data.frame()` if your object was a data frame, `print.factor()` if your object was a factor and so on. That would be super annoying, right? You would always first need to find out what class your object belongs to, and then call the corresponding function - a lot of complicated extra code. Thanks to S3 classes, no matter what class we're dealing with, we

can simply call `print()` - and R will figure out itself which underlying function it needs to call. The S3 system hence allows R functions to behave in different ways for different classes.

15.5.1 Exercises: S3 classes

See Section 18.0.38 for solutions.

1) Write the constructor for a new class "circle". The constructor accepts one argument: a number corresponding to the radius. The creation of the object should fail when you give invalid arguments. Check that your constructor is working by creating some objects of this class!

2) Let's add methods to our class circle. Implement a print method for the circle class. This method should print the radius of the circle object. Test it with your objects.

3) Implement a plot method for the circle class. Hint: use the function `grid.circle()` from the library `grid`.

4) Implement a generic function for class circle that calculates the perimeter (hint: this function should call `UseMethod()`), and the corresponding function with the special S3 naming. In the end, you should be able to call the function like this: `perimeter(myCircle)`. Why do we have to write the generic function here, while we didn't have to do this for `print()`?

5) Following the same principle, implement a generic function for class circle that calculates the area.

6) Check if the two functions above work by calling these functions on some objects of your class!

7) Now, create another class "rectangle". Rectangle should have the same methods as circle. This means, you should be able to create a rectangle object via a constructor (with two arguments, width and height), call `perimeter()` and `area()`, as well as print and plot (use `grid.rect()`) the rectangle object. Always check with examples if your class is working as expected.

8) Say you've got a object "bobby" of a class "mammal":

```
> bobby <- list(name = "bobby", color = "brown", sex = "male")
> class(bobby) <- "mammal"
```

   You now call `print(bobby)`. What do you think will happen - which function will `print()` call? And which function will be called in the end?

9) Define a print method for class "mammal" that prints the name of the mammal, the color and the sex Think first - what will happen if you now call `print(bobby)` again?

10) Our mammal bobby is actually a cow. Assign a second class "cow" to bobby. What do you think will happen when you call `print(bobby)` now?

11) Reverse the order of the class attribute vector of bobby. What do you think will happen when you call `print(bobby)`?

12) We want `print(bobby)` to output "This is bobby. It's a brown male cow". What do you need to implement for this to happen?

Chapter 16

Bonus chapter: tidyverse

If you continue to work in R, you will undoubtedly come across a collection of packages called tidyverse. It's a group of packages that are used to manipulate data and they all use the same underlying design philosophy, grammar, and data structures. There are different schools of thought - some like the user-friendliness of tidyverse and how intuitive its functions are, some don't like how it changes the language structure of R and how it introduces its own syntax. You will make up your own mind about this in the future, but in any case it will be helpful for you to know tidyverse, so you can read and understand scripts of other people using this syntax.

The tidyverse is built up from multiple packages and is still expanding. The most well-known packages are **dplyr** for data manipulation and **ggplot2** for data plotting. **tidyr** is useful for reshaping data (long/wide format).

In this bonus chapter, we will quickly introduce the most common dplyr functions and some other handy tidyverse tricks. If you are interested in a more in-depth look at the topic, there are many free online resources such this one here. The plotting package ggplot is discussed in another bonus chapter.

Of course, you will have to install tidyverse before we begin: `install.packages("tidyverse")`

16.1 dplyr functions for data manipulation

dplyr is the data manipulation package in tidyverse.

Let's load some example data. We will be looking at the `iris` flower dataset. The data is consists of different measurements for individuals of three species of iris.

```
> library(tidyverse)
> library(datasets)
```

Figure 16.1: Tidyverse logo from https://www.tidyverse.org/

```
> data("iris")
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

We will go through the most used functions in dplyr and see the different approaches between base R (meaning, without any packages) and the tidyverse syntax.

16.1.1 `select()`

The `select()` function is used to select specific variables (corresponding to columns) in your dataset. The base R equivalent would be `iris[,"Sepal.Length"]` or `iris$Sepal.Length`.

```
> select(iris, Sepal.Length)
   Sepal.Length
1           5.1
2           4.9
```

```
3           4.7
4           4.6
5           5.0
...
> select(iris, Sepal.Width, Sepal.Length)
    Sepal.Width Sepal.Length
1           3.5          5.1
2           3.0          4.9
3           3.2          4.7
4           3.1          4.6
5           3.6          5.0
...
```

Two things to note:

First, note that in base R, the `" "` are necessary for column names, whereas here, `select(iris, "Sepal.Length")` and `select(iris, Sepal.Length)` both give the same result.

Secondly, even if you select just one column with the `select()` function, it will still output a data.frame, not a vector. This is different from base R. If you are specifically looking for a vector, you can add the `pull()` command. Don't worry about the use of `%>%` yet, it will be explained later.

```
> class(iris[,"Sepal.Length"]) # base R command
[1] "numeric"
> class(select(iris, Sepal.Length)) # select() from dplyr
[1] "data.frame"
> class(select(iris, Sepal.Length) %>% pull())
[1] "numeric"
```

There are some `select_helpers` which allow you to select for very specific cases. You can get an overview on the R help page with `?select_helpers`. Some examples are:

```
> select(iris, ends_with("Length"))
    Sepal.Length Petal.Length
1           5.1          1.4
2           4.9          1.4
3           4.7          1.3
4           4.6          1.5
5           5.0          1.4
...
> select(iris, last_col())
      Species
1      setosa
2      setosa
3      setosa
```

```
4       setosa
5       setosa
...
```

16.1.2 `filter()`

The `filter()` function is used to filter your observations (the rows of your dataset) according to the filters you set. The base R equivalent to this would be something like `iris[iris$Species == "setosa",]`

```
> filter(iris, Species == "setosa")
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5          1.4         0.2  setosa
2           4.9         3.0          1.4         0.2  setosa
3           4.7         3.2          1.3         0.2  setosa
4           4.6         3.1          1.5         0.2  setosa
5           5.0         3.6          1.4         0.2  setosa
...
> filter(iris, Sepal.Length > 7 & Sepal.Width > 3)
  Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1          7.2         3.6          6.1         2.5 virginica
2          7.7         3.8          6.7         2.2 virginica
3          7.2         3.2          6.0         1.8 virginica
4          7.9         3.8          6.4         2.0 virginica
```

The `%in%` operator can be useful to identify if an element belongs to a vector or a column in your data.frame. It can replace the need for multiple `|` statements.

```
> iris[iris$Species == "virginica" | iris$Species == "versicolor",] # without %in%
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
51           7.0         3.2          4.7         1.4 versicolor
52           6.4         3.2          4.5         1.5 versicolor
53           6.9         3.1          4.9         1.5 versicolor
54           5.5         2.3          4.0         1.3 versicolor
55           6.5         2.8          4.6         1.5 versicolor
...
> filter(iris, Species  %in% c("virginica, versicolor")) # with dplyr and %in%
[1] Sepal.Length Sepal.Width  Petal.Length Petal.Width  Species
<0 rows> (or 0-length row.names)
```

16.1.3 'arrange()'

There is a dplyr function called `arrange()` that sorts your data.frame according to the values in a column. In the example below we want our data.frame in descending order of our variable `Sepal.Length`.

```
> arrange(iris, desc(Sepal.Length))
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            7.9         3.8          6.4         2.0  virginica
2            7.7         3.8          6.7         2.2  virginica
3            7.7         2.6          6.9         2.3  virginica
4            7.7         2.8          6.7         2.0  virginica
5            7.7         3.0          6.1         2.3  virginica
...
```

16.1.4 mutate()

The `mutate()` function creates new columns that are functions of existing columns. It can even use a created variable to create other variables within the same call. In this example, the created `Petal.Aspect_ratio` is used in the same call to also create the variable `Petal.Very_oval`.

```
> mutate(iris,
+        Petal.Aspect_ratio = Petal.Length / Petal.Width,
+        Petal.Very_oval = Petal.Aspect_ratio > 6)
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
2            4.9         3.0          1.4         0.2     setosa
3            4.7         3.2          1.3         0.2     setosa
4            4.6         3.1          1.5         0.2     setosa
5            5.0         3.6          1.4         0.2     setosa
...
```

You can also use any arithmetic operators like `+`, `-`, `*`, `^` and so on within `mutate()`.

In all previous examples in this chapter, we never overwrote the original dataset - we merely chose what to display in the console. If you want to save your wrangled dataset to the environment (for example after creating some fancy new columns), you have to explicitly do so. To illustrate this, see the example below.

```
> iris_aspect <- mutate(iris,
+        Petal.Aspect_ratio = Petal.Length / Petal.Width,
+        Petal.Very_oval = Petal.Aspect_ratio > 6)
> head(iris_aspect)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species Petal.Aspect_ratio
1          5.1         3.5          1.4         0.2  setosa               7.00
2          4.9         3.0          1.4         0.2  setosa               7.00
3          4.7         3.2          1.3         0.2  setosa               6.50
4          4.6         3.1          1.5         0.2  setosa               7.50
5          5.0         3.6          1.4         0.2  setosa               7.00
6          5.4         3.9          1.7         0.4  setosa               4.25
  Petal.Very_oval
```

```
1            TRUE
2            TRUE
3            TRUE
4            TRUE
5            TRUE
6           FALSE
```

16.1.5 summarise()

Let us look at some summary statistics of the `Sepal.Length` of our new `iris_aspect` data.frame.

```
> summarise(iris_aspect,
+           mean_Sepal.Length   = mean(Sepal.Length, na.rm = T),
+           sd_Sepal.Length     = sd(Sepal.Length, na.rm = T),
+           n_of_flowers       = n())
  mean_Sepal.Length sd_Sepal.Length n_of_flowers
1          5.843333       0.8280661          150
```

16.1.6 group_by()

These summaries are very useful, but in the iris flower example we have different species we may not want to merge into the same summary statistics. For these cases, dplyr provides the `group_by()` function. You can group your data.frame before you calculate summary statistics.

```
> iris_by_species <- group_by(iris_aspect, Species)
> iris_by_species
# A tibble: 150 x 7
# Groups:   Species [3]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Petal.Aspect_ratio
          <dbl>       <dbl>        <dbl>       <dbl> <fct>                 <dbl>
 1          5.1         3.5          1.4         0.2 setosa                 7
 2          4.9         3            1.4         0.2 setosa                 7
...
```

The generated table is now of the `tibble` class, which is the data.frame equivalent in tidyverse. We're not focusing on this now, as it acts like a data.frame in almost every way. What we want to focus on is that we can see on the top of the table that our data is now grouped by `Species`, and that there are three species in our table. We could ungroup our data with the function `ungroup()`.

Let's see what happens with our grouped data if we use the `summarise()` function from above.

```
> summarise(iris_by_species,
+           mean_Sepal.Length   = mean(Sepal.Length, na.rm = T),
+           sd_Sepal.Length     = sd(Sepal.Length, na.rm = T),
```

```
+           n_of_flowers       = n())
# A tibble: 3 x 4
  Species    mean_Sepal.Length sd_Sepal.Length n_of_flowers
  <fct>                  <dbl>           <dbl>        <int>
1 setosa                  5.01           0.352           50
2 versicolor              5.94           0.516           50
3 virginica               6.59           0.636           50
```

We can also define multiple groups.

```
> iris_by_two <- group_by(iris_aspect, Species, Petal.Very_oval)
> summarise(iris_by_two,
+           mean_Sepal.Length   = mean(Sepal.Length, na.rm = T),
+           sd_Sepal.Length     = sd(Sepal.Length, na.rm = T),
+           n_of_flowers       = n())
# A tibble: 4 x 5
# Groups:   Species [3]
  Species    Petal.Very_oval mean_Sepal.Length sd_Sepal.Length n_of_flowers
  <fct>      <lgl>                       <dbl>           <dbl>        <int>
1 setosa     FALSE                        5.13           0.369           19
2 setosa     TRUE                         4.93           0.326           31
3 versicolor FALSE                        5.94           0.516           50
4 virginica  FALSE                        6.59           0.636           50
```

Hm, interesting, there is only four rows in this summary. Well, it seems that only for the `setosa` species there were both `Petal.Very_oval == TRUE` and `Petal.Very_oval == FALSE`. You can investigate this by looking at the column `n_of_flowers`. For two species, the number is `50`. We know our data well enough to see that this means all of the flowers are accounted for.

16.1.7 Piping

The tidyverse uses a special operator `%>%` for piping. Because it is used so often, it may be worth learning the keyboard shortcut for it:

- `Ctrl`+`Shift`+`M` on Windows and Linux
- `Cmd`+`Shift`+`M` on Mac

It basically means "take my `data` and with it use `function()`".

```
> iris %>% ncol()
[1] 5
```

Because the data.frame is defined before we pipe, we don't have to mention it anymore in the `ncol()` function.

Piping can be done many times in a row:

```
> iris %>% select(Sepal.Width) %>% sum() %>% round(digits = 0)
[1] 459
```

Using the onion principle of many lengthy brackets in base R can make your code very hard to read. Piping is supposed to give your code better readability, especially if you use proper spacing and multiple lines.

To show this, we are again comparing base R to the tidyverse commands:

```
> # base R
> round(sum(iris_aspect$Sepal.Width[iris_aspect$Petal.Aspect_ratio > 5 & iris_aspect$Sp
[1] 115
> # dplyr
> iris_aspect %>%
+   filter(Petal.Aspect_ratio > 5,
+          Species == "setosa") %>%
+   select(Sepal.Width) %>%
+   sum(na.rm = TRUE) %>%
+   round(digits = 0)
[1] 115
```

Using the piping operator %>% all the grouping and summarising we did above can be done in one step.

```
> iris %>%
+   mutate(
+     Petal.Aspect_ratio = Petal.Length / Petal.Width,
+     Petal.Very_oval = Petal.Aspect_ratio > 6) %>%
+   group_by(Species, Petal.Very_oval) %>%
+   summarise(
+     mean_Sepal.Length   = mean(Sepal.Length, na.rm = T),
+     sd_Sepal.Length     = sd(Sepal.Length, na.rm = T),
+     n_of_flowers       = n())
# A tibble: 4 x 5
# Groups:   Species [3]
  Species    Petal.Very_oval mean_Sepal.Length sd_Sepal.Length n_of_flowers
  <fct>      <lgl>                       <dbl>           <dbl>        <int>
1 setosa     FALSE                        5.13           0.369           19
2 setosa     TRUE                         4.93           0.326           31
3 versicolor FALSE                        5.94           0.516           50
4 virginica  FALSE                        6.59           0.636           50
```

16.1.8 Outlook

There are many more packages in the tidyverse universe. This short bonus chapter just gives some insight into the overall topic.

16.1.9 Exercises: Tidyverse

See Section 18.0.39 for solutions.

1. Find the rows where the `Sepal.Length` in our dataset is exactly 5.0, 6.0 or 7.0. Hint: avoid the use of `|`.
2. Load in another example dataset using `data("ChickWeight")`. For every measured time, find the mean weight of all chicks. Exclude diet number 4. Try to use the piping operator.

Chapter 17

Bonus chapter: ggplot2

Looking for plotting examples online, you will most likely find solutions including the ggplot2 package instead of the base R solutions you learned in this class. The package can be either installed directly with `install.packages("ggplot2")` or within another popular package for data manipulation we've just talked about: tidyverse (`install.packages("tidyverse")`).

The ggplot2 syntax follows slightly different rules than the `plot()` function, but feels more intuitive for some users. One of the main differences to base R is, that it does not take vectors as an input. Therefore, your data has to be in a data.frame. This can be an advantage when you load large tables and want to group different aspects without much data manipulation.

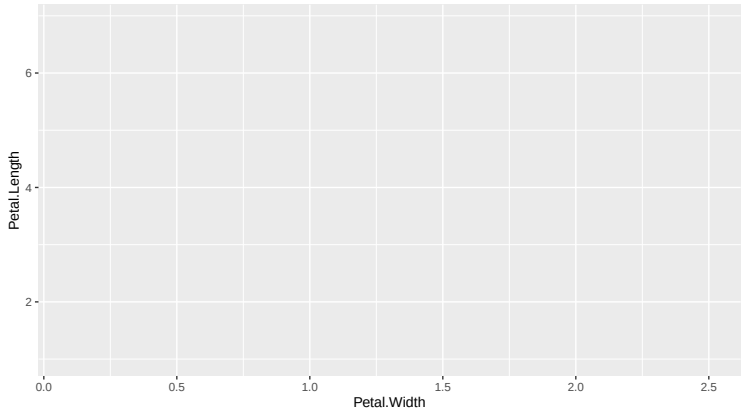
So let's load some example data: the "iris"-flower dataset contains data from 3 species of iris, containing sepal length, sepal width, petal length and petal width (in cm):

```
> library(datasets)
> data("iris")
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

We will now plot the iris sepal length against the sepal width with different shapes and colors for each species.

Just like in base R, we will first define the plotting area:

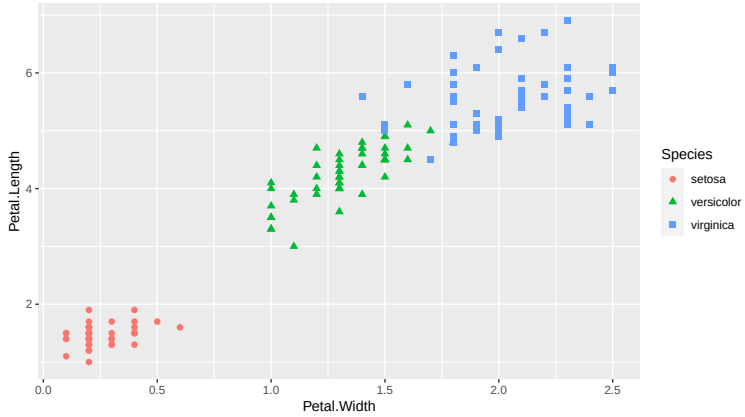
```
> library(ggplot2)
> ggplot(iris, aes(x=Petal.Width, y= Petal.Length))
```



`aes` stands for aesthetics and defines all visual properties that are part of the source dataframe. `Petal.Width` and `Petal.Length` are column names from the data frame.

Now let's add some data points to the plot and define that the color and shape should be defined by the column named "Species":

```
> ggplot(iris, aes(x=Petal.Width, y=Petal.Length)) +
+   geom_point(aes(color=Species, shape=Species), size=2)
```

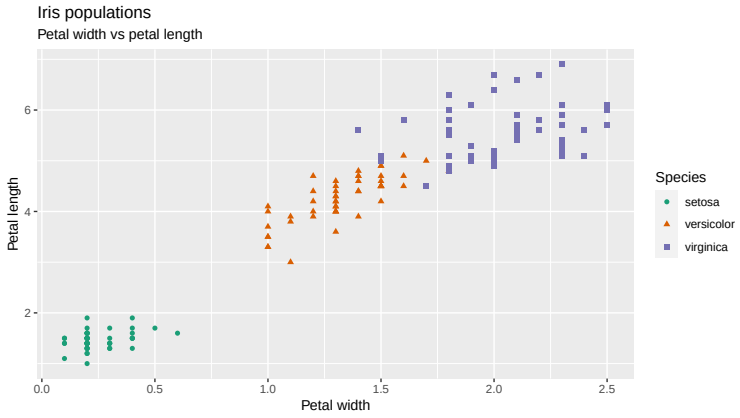


Another difference to base R plotting is, that every element added to the plot has to be added directly to the main plot with a `+`, adding as many layers to the plot as desired. In the above case, `geom_point` draws points. There are many more geom layers that can be drawn like `geom_line` for lines or `geom_bar` for bar charts. You can find a complete list at https://ggplot2.tidyverse.org/reference/ .

Of course we will need to add some titles and rename the axis. The colors can

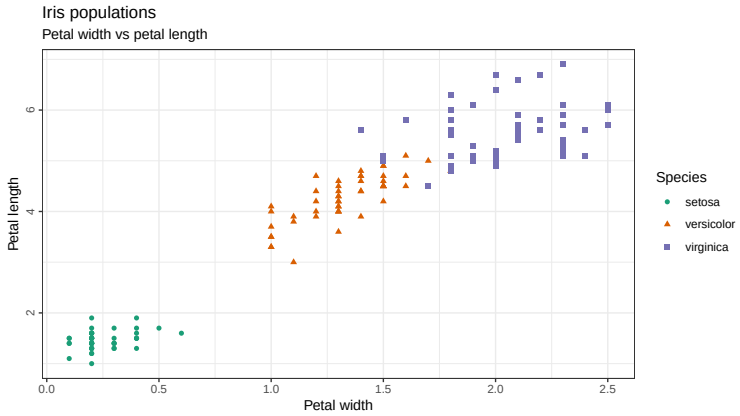
be set manually with `scale_color_manual` or taken from a color brewer palette (`scale_color_brewer`).

```
> ggplot(iris, aes(x=Petal.Width, y=Petal.Length)) +
+   geom_point(aes(color=Species, shape=Species)) +
+   scale_color_brewer(palette="Dark2") +
+   ggtitle("Iris populations", subtitle="Petal width vs petal length") +
+   xlab("Petal width") + ylab("Petal length")
```



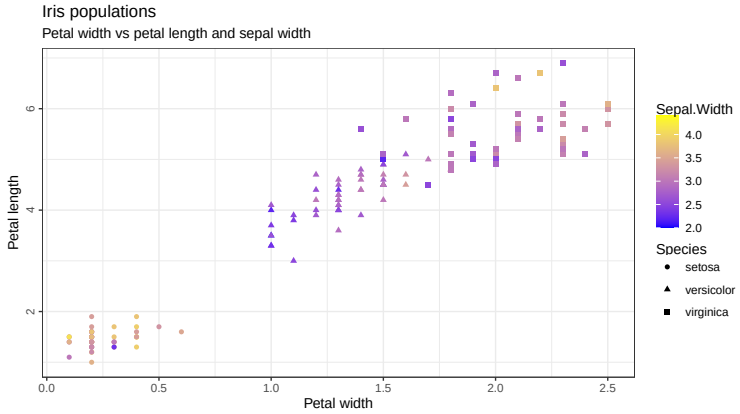
The whole appearance of the plot can be changed by changing the theme of ggplot. There are multiple predefined themes to choose from. Here are some examples but also within the predefined themes everything can be changed and adapted to your needs: `theme_grey()` is the default, `theme_bw()`, `theme_light()`, `theme_dark()`, `theme_minimal()`, `theme_classic()`, `theme_void()`, `theme_test()`. You can play around which theme you like most. Within the theme-argument you can define background colors, gridlines, text sizes, etc.

```
> ggplot(iris, aes(x=Petal.Width, y=Petal.Length)) +
+   geom_point(aes(color=Species, shape=Species)) +
+   scale_color_brewer(palette="Dark2") +
+   ggtitle("Iris populations", subtitle="Petal width vs petal length") +
+   xlab("Petal width") + ylab("Petal length") +
+   theme_bw() +
+   theme(axis.text.y=element_text(angle=90, hjust=0.5)) # turn y-axis text by 90 degrees and adj
```



Maybe we also want to add another column to our plot: The sepal width should be displayed as a gradient color by defining it in the aestethics from geom_point. The same way, stroke, shape and size can be used to discriminate groups within the dataset. The colors can be customized with the `scale_color` argument.

```
> ggplot(iris, aes(x=Petal.Width, y=Petal.Length)) +
+   geom_point(aes(color=Sepal.Width, shape=Species)) +
+   scale_color_gradient(low="blue", high="yellow") +
+   ggtitle("Iris populations", subtitle="Petal width vs petal length and sepal width")
+   xlab("Petal width") + ylab("Petal length") +
+   theme_bw() +
+   theme(axis.text.y=element_text(angle=90, hjust=0.5),
+                      legend.spacing.y = unit(0, "cm")) # decrease space between the
```



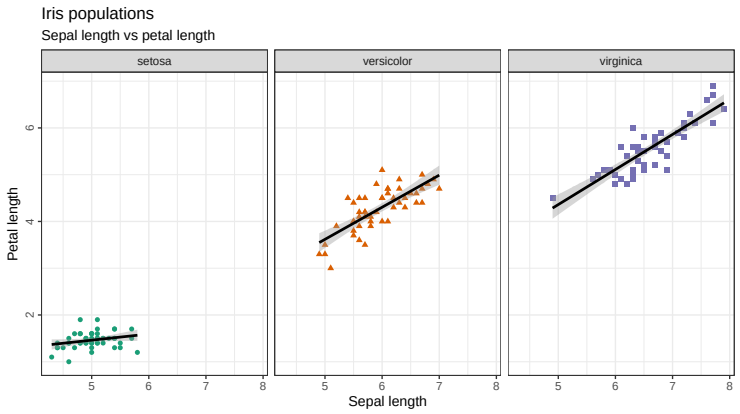
A ggplot can also be saved in a variable. The plot will then only be drawn once the variable is being called. Let's save this scatter-plot for a later use:

```
> scatterPlot <- ggplot(iris, aes(x=Petal.Width, y=Petal.Length)) +
+                    geom_point(aes(color=Sepal.Width, shape=Species)) +
+                    scale_color_gradient(low="blue", high="yellow") +
```

```
+                     ggtitle("Petal width vs petal length and sepal width") +
+                     xlab("Petal width") + ylab("Petal length") +
+                     theme_bw() +
+                     theme(axis.text.y=element_text(angle=90, hjust=0.5), legend.spacing.y = uni
```

The three species could also be plotted in separate plots as a facet-grid and we could also add a smooth line. Let's do this for a comparison of sepal length and petal length:

```
> ggplot(iris, aes(x=Sepal.Length, y=Petal.Length)) +
+   geom_point(aes(color=Species, shape=Species)) +
+   scale_color_brewer(palette="Dark2") +
+   ggtitle("Iris populations", subtitle="Sepal length vs petal length") +
+   xlab("Sepal length") + ylab("Petal length") +
+   facet_wrap(~Species) + #create one plot per species
+   geom_smooth(aes(group=Species), method=lm, formula='y ~ x', color='black') +  #adding linear
+   theme_bw() +
+   theme(axis.text.y=element_text(angle=90, hjust=0.5),
+                      legend.position = "none") # suppress legend
```



The `method=lm` argument is plotting a linear model to your data-points. You can fit different models to it like `glm` for generalized linear regression or `loess` for local regression or even a function you defined yourself. If you leave the argument out, the method is chosen by the program automatically.

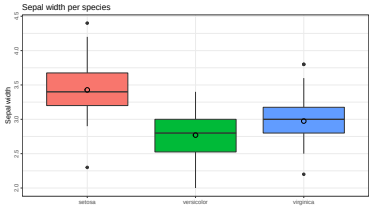
Let's save this scatter-plot for a later use:

```
> facetPlot <- ggplot(iris, aes(x=Sepal.Length, y=Petal.Length)) +
+                     geom_point(aes(color=Species, shape=Species)) +
+                     scale_color_brewer(palette="Dark2") +
+                     ggtitle("Iris populations", subtitle="Sepal length vs petal length") +
+                     xlab("Sepal length") + ylab("Petal length") +
+                     facet_wrap(~Species) + #create one plot per species
+                     geom_smooth(aes(group=Species), method=lm, formula='y ~ x', color='black') +
```

```
+                   theme_bw() +
+                   theme(axis.text.y=element_text(angle=90, hjust=0.5),
+                                       legend.position = "none") # suppress legend
```

To draw a grouped boxplot for all sepal widths grouped per species we simply use the `geom_boxplot` function. We can also add the mean with the `stat_summary`:

```
> ggplot(iris, aes(x=Species, y=Sepal.Width)) +
+           geom_boxplot(aes(fill=Species)) +
+           ggtitle("Sepal width per species") + ylab("Sepal width") +
+           guides(fill="none") + # suppressing legend
+           stat_summary(fun=mean, geom="point", shape=1, size=3)+ #calculate the mean
+           theme_bw()+ theme(axis.text.y=element_text(angle=90, hjust=0.5),
+                             axis.title.x=element_blank()) #supressing x-axis title
```



Saving this plot in a variable as well...

```
> boxPlot <- ggplot(iris, aes(x=Species, y=Sepal.Width)) +
+               geom_boxplot(aes(fill=Species)) +
+               ggtitle("Sepal width per species") + ylab("Sepal width") +
+               guides(fill="none") + # supressing legend
+               stat_summary(fun=mean, geom="point", shape=1, size=3) +
+               theme_bw() +
+               theme(axis.text.y=element_text(angle=90, hjust=0.5),
+                                   axis.title.x=element_blank()) #supressing x-axis ti
```

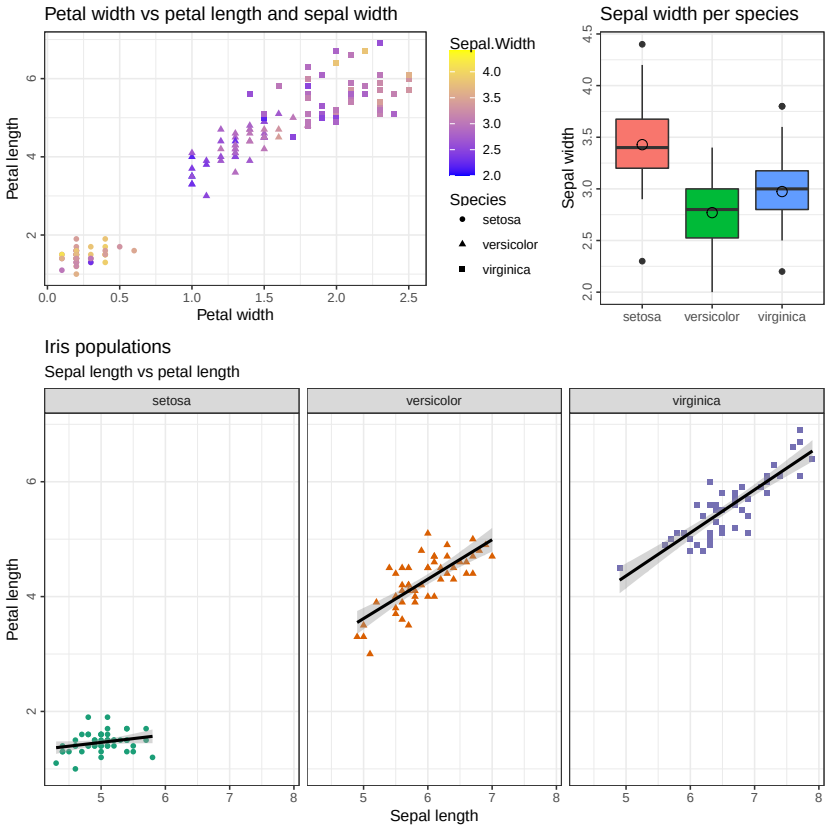
With the additional package `gridExtra` (if it is not already installed, install with `install.packages("gridExtra")`, we can now combine the three plots from above:

```
> library(gridExtra)

Attaching package: 'gridExtra'
The following object is masked from 'package:dplyr':

    combine
> grid.arrange(scatterPlot, boxPlot, facetPlot,nrow = 2, ncol=2,
+              widths = c(2, 1),
+              heights = c(1,1.5),
+              layout_matrix = rbind(c(1, 2),
```

```
+                                          c(3, 3)))
```



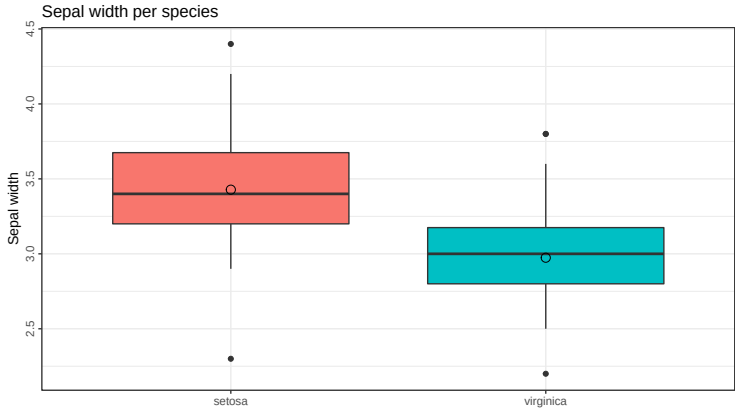
The `nrow` and `ncol` stand for the number of rows and columns in this plot. The `heights` and `widths` parameters define, how the rows and columns should be distributed. E.g. `widths = c(2, 1)` means that the first column should be twice the size of the second. The `layout_matrix` defines how the plots should be arranged, where the numbers represent the plots in the order they have been called. So in this case, the first row contains the plots number 1 (scatterPlot) and 2 (boxPlot) in a ratio 2:1, and the second row contains the plot 3 (facetPlot) over both columns. The second row is 1.5 times the height of the first row.

17.0.1 Combining ggplot2 with other tidyverse syntax

You can take advantage of the synergy between ggplot2 and tidyverse. You can wrangle your data and then pipe it into ggplot2 all in one step.

```
> iris %>%
+   mutate() %>%
+   filter(Species != "versicolor") %>%
+   ggplot(aes(x=Species, y=Sepal.Width)) +
```

```
+                 geom_boxplot(aes(fill=Species)) +
+                 ggtitle("Sepal width per species") + ylab("Sepal width") +
+                 guides(fill="none") + # supressing legend
+                 stat_summary(fun=mean, geom="point", shape=1, size=3) +
+                 theme_bw() +
+                 theme(axis.text.y=element_text(angle=90, hjust=0.5),
+                                   axis.title.x=element_blank()) #supressing x-axis ti
```



17.0.2 Exercises: ggplot2

See Section 18.0.40 for solutions.

1. Load the dataset "CO2" from the datasets-library. It contains the following columns: Plant: unique identifier for each plant Type: origin of the plant Treatment: a factor with levels "nonchilled" and "chilled" conc: carbon dioxide concentrations (mL/L) uptake: carbon dioxide uptake rates (µmol/mˆ2 sec)

Plot the CO2-concentration (x) against the CO2-uptake (y) in a scatter-plot. The points should be colored by the origin of the plant. Add a title and axis-labels.

1. Use `facet_wrap` to group the plot from exercise 1 by Treatment and add a smoothing line (no model specified) for each treatment.

Chapter 18

Solutions to Exercises

18.0.1 Arithmetic Operators

Corresponding exercises: see Section 2.0.1

1. `1+2+3`
2. `1/(1+2)`
3. `2^(1+2)`

18.0.2 Functions

Corresponding exercises: see Section 2.1.2

1. Type `?round` or `help(round)` into the console. Purpose: rounds the values in its first argument to the specified number of decimal places. Arguments: 1) x = number that should be rounded. 2) digits = how many decimal places (how many numbers after the comma should remain). Default: the second argument (digits) has default 0.
2. ```
   round(1.23456789)
   round(1.23456789, 3)
   round(1.23456789, 7)
   ```
3. ```
   5.7
   5.7
   5.7
   1
   ```
4. Searching in the help pages does not help much. Try googling something

like “R binomial coefficient”.
`choose(5,3)`

18.0.3 Variables

Corresponding exercises: see Section 2.2.2

1. ```
   value_1 <- 6.7
   value_2 <- 56.3
   ```

2. ```
   sqrt(value_1)
   sqrt(value_2)
   ```

3. ```
   x <- ((2*value_1)/value_2)+(value_1*value_2)
   ```

18.0.4 R-Scripts

Corresponding exercises: see Section 3.0.1

1. ```
   #--------------------------------
   # Experimenting with logarithms
   #--------------------------------

   # the data
   my_data <- pi

   # different bases of the logarithm
   base_1 <- 10
   base_2 <- exp(1)

   # print the logarithm of my_data for the different bases
   print(log(my_data, base=base_1))
   print(log(my_data, base=base_2))
   ```

2. Execute your script using the buttons “Run” and “Source”, or the corresponding shortcuts `Ctrl` + `Enter` and `Ctrl`+ `Shift`+ `Enter`, respectively.

18.0.5 Introduction to vectors

Corresponding exercises: see Section 4.0.4

1. ```
   x <- c(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5)
   ```

2. ```
   c(-6, x, 6)
   ```

3. ```
   vec1 <- c(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5)
   vec2 <- c(6, 7, 8, 9, 10)
   vec3 <- c(vec1, vec2)
   print(vec3)
   length(vec3)
   ```

4. ```
   -5:5
   10:6
   ```

5. ```
   vec1 <- rep(1:5, times = 3)
   vec2 <- rep(1:5, each = 3)
   vec3 <- rep(1:5, times = 1:5)
   vec4 <- rep(1:5, each = 3, times = 2)
   ```

6. ```
   vec1 <- seq(0, 5, by=0.5)
   vec2 <- seq(1, 8, length.out=6)
   vec3 <- seq(10, -0.4, by=-0.8)
   ```

7. ```
   x <- -5:5
   x[8]
   ```

8. ```
   x[c(2,4,6,8,10)]
   # or
   x[seq(2,10,2)]
   ```

9. ```
   x[9:11] <- c(30, 40, 50)
   # or
   x[(length(x)-2):length(x)] <- c(30, 40, 50)
   ```

10. ```
    x <- numeric(5)
    x[1:5] <- 1:5
    ```

11. ```
    x <- -5:5
    x[c(-2,-4,-6,-8,-10)]
    # or
    x[seq(-2, -10, -2)]
    ```

18.0.6 Numeric vectors

Corresponding exercises: see Section 4.1.2

1. ```
   a <- -5:5
   b <- 10:0
   ```

```
a-b
a+b
a*b
```

2. ```
   sum(c(a, b))
   ```

3. ```
   min(a)
   max(a)
   mean(a)
   ```

4. ```
   (a - mean(a)) / sd(a)
   ```

5. ```
   a * c(1, 1, 5)
   # You get a warning message because recycling expects that the longer vector is a
   ```

18.0.7 Logical vectors

Corresponding exercises: see Section 4.2.1

1. ```
   sum(y > x)
   ```

2. ```
   x[x %% 2 == 0] = 1
   ```

3. ```
   x[x >= 0 & x %% 2 == 1]
   ```

4. ```
   x[x < -10 | x > 13]
   ```

5. ```
   x[rep(c(F, F, F, T), length.out=length(x))] <- 1
   ```

18.0.8 Character vectors

Corresponding exercises: see Section 4.3.1

1. ```
   paste0(c("A", "B"), 1:100, collapse = " ")
   ```

2. ```
   paste(paste0("A", 1:100), paste0("B", 1:100), collapse = " ")
   ```

3. ```
   paste(hello, world, bang)
   paste0(hello, world, bang)
   paste0(paste(hello, world), bang)
   paste0(paste(hello, world, sep = "-"), bang)
   paste0(paste(hello, world), paste0(rep(bang, 3), collapse = ""))
   paste0(paste(hello, world), rep(bang, 3), collapse = " ")
   ```

18.0.9 Factors

Corresponding exercises: see Section 4.4.1

1.
```
Levels: 3 4 5 7 8
```

2.
```
months <- c("july", "december", "july", "july", "may", "december", "july", "september", "may
fMonths <- factor(months)
print(fMonths)
str(fMonths)
# There are 6 levels: august december july may october september.
```

18.0.10 Sorting and shuffling vectors

Corresponding exercises: see Section 4.6.1

1.
```
x <- c(3, 4, -6, 2, -7, 2, -1, 0)
sort(x)
sort(x, decreasing = T)
```

2.
```
x <- c(3, 4, -6, 2, -7, 2, -1, 0)
sort(x[1:3])
```

3.
```
v <- seq(8,37,length.out=98)
V <- sample(v,10)
sort(V, decreasing = TRUE)
```

18.0.11 Creating matrices

Corresponding exercises: see Section 5.1.1

1.
```
matrix(1, nrow=5, ncol=10)
```

2.
```
diag(1:2, nrow=10)
```

3.
```
A <- matrix(c(1:10, 10:1), nrow=2, byrow=TRUE)
```

4.
```
A <- rbind(1:10,10:1)
```

5.
```
b <- 1:80
B <- matrix(b, ncol=5)
k <- dim(A)[1]
```

6.
```
A <- matrix(2, ncol=3, nrow=3)
diag(A) <- 1
```

18.0.12 Accessing Matrix Elements

Corresponding exercises: see Section 5.2.1

1.
```
A <- matrix(1:16, nrow=4, byrow=TRUE); B <- A[-4,]
```

2.
```
sum(A[2,])
sum(A[,3])
```

3.
```
# The trick is to use the
A[which(A + 1 < (A/3)^2)] <- -1
```

18.0.13 Matrix Arithmetic

Corresponding exercises: see Section 5.3.1

1.
```
Z <- matrix(c(-1,1,1,-1), nrow=2); Z %*% t(Z)
```

2.
```
matrix(1:4, nrow=2, byrow=TRUE) %*% matrix(c(-1,1), nrow=2, ncol=3, byrow=TRUE)
```

3.
```
solve(matrix(c(1,-2,-1,-3,0,1,7,1,0), nrow=3))
```

4.
```
E <- matrix(1:4, ncol = 2, nrow = 2)
G <- matrix(5:8, ncol = 2, nrow = 2)
E
G
E * G
E / G
E %*% G
E + G
E - G
E == G
```

18.0.14 Data Frames

Corresponding exercises: see Section 6.1.1

1.
```
x <- c(5:0);
y <- c(0:5);
d <- data.frame (x,y);
```

2. ```
   w <- c(10:6);
   z <- c(6:10);
   e <- data.frame (z,w);
   names(e)<-names(d);
   d <- rbind(d,e)
   ```

3. ```
   d <-cbind(d,compared=c(x>y,z>w))
   ```

18.0.15 Accessing data frames

Corresponding exercises: see Section 6.2.1

1. ```
   d <- data.frame(x=1:10,y=c(1,10,-2,3,8,-7,2,1,9,4));
   d[d$x==d$y,]
   ```

2. ```
   mean((d$y/d$x)[(length(d$x)-4):length(d$x)])
   ```

18.0.16 Manipulating data frames

Corresponding exercises: see Section 6.3.1

1. ```
   id <- paste(c("X"),1:10,sep="")
   x <- sample(-10:10, 10, replace=TRUE)
   y <- sample(-10:10, 10, replace=TRUE)
   data <- data.frame(id,x,y)
   data$y <- data$y^2
   ```

2. ```
   data <- cbind(data, ok = data$y > 20)
   other <- data[which(data$ok & x>3), c("id", "y")]
   ```

18.0.17 Lists

Corresponding exercises: see Section 7.3.1

1. ```
   holiday <- list(country=c("CH", "USA", "Japan"), continent=c("Europe", "The Americas", "East
   ```

2. ```
   x <- list(holiday=holiday, place=c("mountain", "beach"), day=as.numeric(1:20), like=c(TRUE,F
   ```

3. ```
    x[[1]][[1]][2]
   ```

4. ```
   # There are two solutions:
   x$holiday[[2]][1]
   ```

```r
# or
x[[1]]$continent[1]
```

5.
```r
x[[c(1,1,3)]]
```

6.
```r
name <- "day"; x[[name]]
```

7.
```r
x[c(2,4)]
```

8.
```r
Chapters <- c("Chapter 3. R-Scripts","Chapter 4. Vectors","Chapter 5. Matrices", '
strsplit(Chapters,"\\. ")[[c(4,2)]]
```

18.0.18 Conditionals

Corresponding exercises: see Section 8.1.1

1.
```r
x <- c(1,4,3)
if(x[1]<x[2]&&x[2]<x[3]){ print("increasing") }else{ print("not increasing") }
```

18.0.19 Loops

Corresponding exercises: see Section 8.2.3

1.
```r
x <- sample(10)
for(i in 2:length(x)){
if(x[i] < x[i-1]){
    print("Vector is not strictly increasing!");
  }
}
```

2.
```r
v <- c()
for(i in 1:10){
  for(j in 1:i){
     v <- c(v, i)
  }
}
# Note that the inner loop could be replaced by `rep(i, i)`.
```

3.
```r
m <- matrix(1:120, nrow=3)
v <- integer(dim(m)[2])
for(i in 1:dim(m)[2]){
  v[i] <- sum(m[,i])
}
```

4.
```r
x <- 1
while(x <= 10){
  print(paste("I like number", x))
  x <- x + 1
}
```

5.
```r
F <- numeric(100)
F[1] <- 0
F[2] <- 1
for(i in 3:100){
  F[i] <- F[i-1] + F[i-2]
}
```

6.
```r
x <- sample(100); y <- sample(100)
for(i in 1:length(x)){
  if(x[i] > y[i]){
    print(x[i]*y[i])
  } else if(x[i] < y[i]){
    print(x[i]+y[i])
  }
}
```

7.
```r
a <- c(5,6,8,2,3,7,8,1,2,3,4)
i <- 1
while(a[i] >= 2){
  print(a[i])
  i <- i + 1
}
```

8.
```r
a <- matrix(data=NA,nrow=3,ncol=3)
for(i in 1:nrow(a)){
  for(j in 1:ncol(a)){
      a[i,j] <- i+j
   }
}
```

9.
```r
i <- 1; s <- i;
while(s < 5000000){
  i <- i + 1
  s <- s * i
}
print(i)
```

18.0.20 Basic Plotting

Corresponding exercises: see Section 9.0.2

1.
```
x <- seq(-5, 5, length.out=20)
y <- x^2*2
plot(x, y, type='b', col='red', pch=19, lty=2, xlab="My variable x", ylab="Some tr
```

2.
```
par(mar=c(2,2,0,0))
plot(x, y, type='l', col=colors()[sample(length(colors()), 1)], lwd=10, xlab="", y
```

3.
```
x <- rep(1:26, each=26)
y <- rep(1:26, 26)
plot(x, y, pch=15, col=colors(), cex=1.6)
```

4.
```
#There are at least two solutions:
plot(1:100, pch=19, col=c(rep("orange", 6), "red"))
# or
plot(1:100, pch=19, col=c("orange", "red")[1 + (1:100 %% 7 == 0)])
```

18.0.21 Plotting

Corresponding exercises: see Section 9.1.5

1.
```
x <- seq(1, 100, by=0.5)
d <- data.frame(x=x, y=sqrt(x), z=log(x))
plot(d$x, d$y, type='l', col='red', xlab="x", ylab="A transformation of x")
lines(d$x, d$z, col='blue', lty=2)
legend('bottomright', legend=c("sqrt(x)", "log(x)"), col=c('red', 'blue'), lty=c(1
```

2.
```
plot(0, type='n', xlim=c(-10, 10), ylim=c(-1,1), xlab="x", ylab="y")
abline(v=0, col='red', lty=1)
abline(h=0, col='red', lty=1)
text(c(5, -5, -5, 5), c(0.5, 0.5, -0.5, -0.5), labels=c("I", "II", "III", "IV"))
```

3.
```
av.high <- c(32, 34, 34, 32, 32, 30, 32, 30, 30, 30, 32, 32)
av.rain <- c(16, 31, 104, 131, 161, 155, 193, 225, 192, 199, 76, 27)
par(las=1, mar=c(4,4,0.5,4)) # las changes the orientation of the labels
# of the axis. The plot becomes more readable this way
plot(1:12, av.high, ylim=c(0, max(av.high)), type='b', col='red', pch=19, xlab="Mo
scale <- max(av.rain) / max(av.high)
lines(1:12, av.rain / scale, type='b', col='blue', pch=17)
labels <- pretty(av.rain)
axis(side=4, at=labels / scale, labels=labels)
mtext("Average rainfall [mm]", side=4, line=3, las=3)
```

```
legend('bottom', bty='n', col=c('red', 'blue'), lwd=1, pch=c(19, 17), legend=c("Average temp
```

18.0.22 Plotting Multiple Pannels

Corresponding exercises: see Section 9.2.1

1.
```
x <- seq(1, 100, by=0.5)
d <- data.frame(x=x, y=sqrt(x), z=log(x))
par(mfrow=c(1,2))
plot(d$x, d$y, type='l', col='red', xlab="x", ylab="A transformation of x", main="sqrt(x)")
plot(d$x, d$z, col='blue', lty=2, xlab="x", ylab="A transformation of x", main="log(x)")
```

2.
```
x <- 0:100
bases <- 2:13
par(mfrow=c(3,4))
for(i in 1:length(bases)){
  plot(x, log(x, base=bases[i]), col=colors()[i*5], pch=i, xlab="x", ylab="log(x)")
  text(par("usr")[1], par("usr")[3] + 0.1*(par("usr")[4]-par("usr")[3]), paste("base =", bas
}
```

3.
```
layout(matrix(c(1,1,2,3,4,4), nrow = 3, ncol = 2, byrow = TRUE))
Colors <- c("yellow","orange","blue","black")
for(i in 1:4){
 plot(x,a,pch=16,col=Colors[i])
}
```

18.0.23 Probability distributions

Corresponding exercises: see Section 10.0.4

1.
```
dexp(0.15, 2)
```

2.
```
dpois(3, 2)
```

3.
```
qnorm(0.99, mean=5, sd=4)
```

4.
```
dbeta(0.72, 0.7, 0.7)
```

5.
```
qnorm(1-0.3)
```

6.
```
rpois(100, lambda=2)
```

7.
```r
rexp(13, rate=0.5)
```

8.
```r
x <- rnorm(1000)
sum(x < -1)/length(x)
# theoretical expectation
pnorm(-1)
# with more random draws, we get closer to theoretical expectation
x <- rnorm(100000)
sum(x < -1)/length(x)
```

9.
```r
x <- rbinom(1000, size = 10, prob = 0.1)
sum(x > 2)/length(x)
# theoretical expectation: take inverse (we are interested in >2)
1 - pbinom(2, size = 10, prob = 0.1)
```

10.
```r
height <- seq(140, 200, length.out = 1000)
females <- dnorm(height, 164.7, 7.1)
males <- dnorm(height, 178.4, 7.6)

plot(height, females, col = 'dodgerblue', type = "l", ylab = "density")
lines(height, males, col = 'orange2')

myHeight <- 175
abline(v = myHeight, col = 'dodgerblue', lty = 2)
dnorm(myHeight, 164.7, 7.1) # for females
```

11.
```r
height <- seq(140, 200, length.out = 1000)
females <- pnorm(height, 164.7, 7.1)
males <- pnorm(height, 178.4, 7.6)

plot(height, females, col = 'dodgerblue', type = "l", ylab = "cumulative density")
lines(height, males, col = 'orange2')

myHeight <- 175
abline(v = myHeight, col = 'dodgerblue', lty = 2)
pnorm(myHeight, 164.7, 7.1) # for females
```

12.
```r
quantiles <- c(0.05, 0.32, 0.5, 0.68, 0.95)
femalesQuantiles <- qnorm(quantiles, 164.7, 7.1)
malesQuantiles <- qnorm(quantiles, 178.4, 7.6)

# plot probability density function
height <- seq(140, 200, length.out = 1000)
```

```r
females <- dnorm(height, 164.7, 7.1)
males <- dnorm(height, 178.4, 7.6)

plot(height, females, col = 'dodgerblue', type = "l", ylab = "density")
lines(height, males, col = 'orange2')

# add lines
abline(v = femalesQuantiles, col = 'dodgerblue', lty = 4)
abline(v = malesQuantiles, col = 'orange2', lty = 4)
```

13.
```r
lambda <- c(0.5, 1, 1.5)
color <- c("orange", "purple", "cadetblue")
par(mfrow=c(2,1), mar=c(4,4,1,0.1))
x <- seq(0, 5, length.out=100)
# plot probability density function
plot(0, type='n', xlim=range(x), ylim=c(0, 1.5), xlab="x", ylab="P(x)")
for(l in 1:length(lambda)){
  lines(x, dexp(x, rate=lambda[l]), col=color[l])
}
legend('topright', bty='n', col=color, lwd=1, legend=lambda)
# plot cumulative distribution function
plot(0, type='n', xlim=range(x), ylim=c(0, 1), xlab="x", ylab="P(X<=x)")
for(l in 1:length(lambda)){
  lines(x, pexp(x, rate=lambda[l]), col=color[l])
}
legend('bottomright', bty='n', col=color, lwd=1, legend=lambda)
```

14.
```r
x <- numeric(100)
for(i in 2:100){
  x[i] <- rnorm(1, x[i-1], 1)
}
plot(x, type='b', pch=19, col='purple', xlab="index", ylab="x")
```

15.
```r
nRep <- 10000
net <- numeric(nRep)
for(r in 1:nRep){
  # start of one game
  invest <- 1
  expenses <- invest
  # simulate first round
  win <- runif(1) < 18/37
  # simulate all subsequent rounds
  while(!win & invest <= 50){
      invest <- 2 * invest
```

```
      expenses <- expenses + invest
      win <- runif(1) < 18/37
  }
  if(win){
    # we won -> invested money is doubled
    net[r] <- 2 * invest - expenses
  } else {
    # we lost -> no income, only expenses
    net[r] <- 0 - expenses
  }
}
sum(net>0)/length(net)
mean(net)

# The net benefit is very likely to be negative, since the probability of winning
```

18.0.24 Empirical Distributions

Corresponding exercises: see Section 10.1.3

1. ```
   x <- rbinom(1000, size=5, prob=0.2)
   par(mfrow=c(1,2))
   hist(x)
   plot(density(x))
   # the histogram represets the discrete nature of the data
   ```

2. ```
   x <- seq(0, 20, length.out=100)
   plot(x, dnorm(x, mean=10, sd=2), col='red', type='l')
   n <- 10^(1:4)
   for(i in 1:length(n)){
     lines(density(rnorm(n[i], mean=10, sd=2)), lty=i)
   }
   legend('topleft', lty=1:length(n), legend=n)
   ```

3. ```
   data("iris")

   # plot histograms
   par(mfrow = c(1,3))
   hist(iris$Petal.Length[iris$Species == "setosa"], col = "dodgerblue", xlab = "peta
   hist(iris$Petal.Length[iris$Species == "versicolor"], col = "orange2", xlab = "pet
   hist(iris$Petal.Length[iris$Species == "virginica"], col = "firebrick", xlab = "pe

   # plot kernel
   par(mfrow = c(1,1))
   ```
   ```

```
plot(density(iris$Petal.Length), ylim = c(0, 2.6), main = "Distribution of petal length")
lines(density(iris$Petal.Length[iris$Species == "setosa"]), col = "dodgerblue")
lines(density(iris$Petal.Length[iris$Species == "versicolor"]), col = "orange2")
lines(density(iris$Petal.Length[iris$Species == "virginica"]), col = "firebrick")
legend("topright", legend = c("global", "setosa", "versicolor", "virginica"), col = c("black
```

18.0.25 Data Input and Export

Corresponding exercises: see Section 11.0.1

1. ```
   getwd()
   ```

2. Create a directory using any means your file system offers. Then switch there using `setwd()` or with RStudio. Check where you are with `getwd()`.

18.0.26 Exporting and Importing

Corresponding exercises: see Section 11.1.5

1. ```
   d <- data.frame(x=rnorm(100), y=rnorm(100), z=rnorm(100))
   write.table(d, file="my_data_frame.txt", row.names=FALSE)
   d <- read.table("my_data_frame.txt", header=TRUE)
   ```

2. ```
   read.csv2("your_file.txt")
   ```

3. ```
   d <- read.csv2(url("https://www.bfs.admin.ch/bfsstatic/dam/assets/12647632/appendix"), strin
   d <- d[-1,]
   plot(d$pop1930, d$pop2018, xlab="1930", ylab="2018")
   ```

18.0.27 Listing files

Corresponding exercises: see Section 11.2.1

1. ```
   length(list.files(path="~/Downloads",pattern="*.pdf"))/length(list.files(path="~/Downloads")
   ```

18.0.28 Exporting graphics

Corresponding exercises: see Section 11.3.1

1. ```
   x <- rnorm(100)
   y <- 4 + 1.5*x + rnorm(100, sd=sqrt(0.2))
   pdf("x_vs_y.pdf")
   par(mar=c(4,4,0.1,0.1))
   plot(x, y)
   dev.off()
   ```

2.
```
pdf("x_vs_y.pdf", width=2, height=2)
par(mar=c(4,4,0.1,0.1))
plot(x, y)
dev.off()
```

3.
```
jpeg("x_vs_y_low.jpeg", width=480, height=480, pointsize=12)
par(mar=c(4,4,0.1,0.1))
plot(x, y)
dev.off()
jpeg("x_vs_y_high.jpeg", width=1280, height=1280, pointsize=12*1280/480)
par(mar=c(4,4,0.1,0.1))
plot(x, y)
dev.off()
```

18.0.29 RStudio Workspace

Corresponding exercises: see Section 11.4.1

18.0.30 Installing packages

Corresponding exercises: see Section 12.0.1

1.
```
install.packages("RColorBrewer")
library(RColorBrewer)
x <- seq(-4, 4, by = 0.1)
numSds <- 10
cols <- brewer.pal(numSds, "Spectral")
# generate empty plot
plot(0, type = "n", xlim=range(x), ylim=c(0,4), xlab="x", ylab = "density")
for (i in 1:numSds){
  y <- dnorm(x, 0, i/numSds)
  lines(x, y, col = cols[i])
}
```

18.0.31 Writing Functions

Corresponding exercises: see Section 13.0.1

1.
```
square <- function(x){
  return(x*x)
}
```

2.
```
convertFahrenheitToCelsius <- function(temp_F) {
  temp_C <- (temp_F - 32) * 5/9
```

```
  return(temp_C)
}
```

3. ```
   calculateSampleVariance <- function(x){
     n <- length(x)
     res <- 1/(n-1) * sum((x - mean(x))^2)
     return(res)
   }
   calculateSkew <- function(x){
     n <- length(x)
     nominator <- 1/(n-2) * sum((x - mean(x))^3)
     denominator <- var(x)^(3/2)
     return(nominator / denominator)
   }
   ```

4. ```
   calculateBothNA <- function(x, y){
     return(sum(is.na(x) & is.na(y)))
   }
   ```

18.0.32 Input to functions

Corresponding exercises: see Section 13.1.3

1. ```
   takePower <- function(x, y = 2){
     return(x^y)
   }
   ```

18.0.33 Checking values

Corresponding exercises: see Section 13.2.1

1. ```
   convertKelvinToFahrenheit <- function(temp_K) {
     if (temp_K < 0) stop("Kelvin can not have negative values!")
     temp_F <- temp_K * 9/5 - 459.67
     return(temp_F)
   }
   ```

18.0.34 Return Values

Corresponding exercises: see Section 13.3.4

1. ```
   computeFactorial <- function(n){
     if (n == 0)
       return(1)
     else {
   ```
   ```

```
    prod <- 1
    for (p in 1:n){
      prod <- p * prod
    }
    return(prod)
  }
}
```

2.
```
computeFactorial <- function(n){
  if (n == 0)
    return(1)
  else
    return(prod(1:n))
}
```

3.
```
simulateDiceNum6 <- function(){
  # first simulate
  simul <- sample(1:6, 100, replace=T)
  # now count number of 6's
  num6 <- 0
  for (i in simul){
    if (i == 6)
      num6 <- num6 + 1
  }
  return(list(simulation = simul, num6 = sum))
}
```

4.
```
simulateDiceNum6 <- function(){
  # first simulate
  simul = sample(1:6, 100, replace=T)
  # now count number of 6's
  num6 <- sum(simul == 6)
  return(list(simulation = simul, num6 = sum))
}
```

5.
```
fizzbuzz <- function(x){
  if (x %% 3 == 0 & x %% 5 == 0)
    return("fizzbuzz")
  else if (x %% 3 == 0)
    return("fizz")
  else if (x %% 5 == 0)
    return("buzz")
  else return(x)
}
```

18.0.35 Environment

Corresponding exercises: see Section 13.4.1

1. `10`

18.0.36 Loops are slow

Corresponding exercises: see Section 14.0.2

1. ```
   x <- runif(10^6)
   system.time(exp(x))/system.time(x+x)
   ```

2. ```
   mySum <- function(x){
     s <- 0
     for(i in 1:length(x)){
       s <- s + x[i];
     }
     return(s)
   }
   system.time(mySum(x))
   system.time(sum(x))
   ```

3. ```
   myExpSum <- function(x){
     s <- 0
     for(i in 1:length(x)){
       s <- s + exp(x[i])
     }
     return(s)
   }
   system.time(myExpSum(x))

   # As you will notice, calculating the sum of exp(x) takes about twice as long as calculating
   # When using functions on vectors (first exercise), it took about 4 times as long.
   # This suggests, that the majority of the calculation time is now caused by the for loop, no
   ```

4. ```
   m <- matrix(rpois(100*10, lambda=rep(1:10, each=100)), ncol=10)
   ```

18.0.37 Apply

Corresponding exercises: see Section 14.1.6

1. ```
   lapply(sample(1:1000, size=100, replace=TRUE), rep, 10)
   ```
   ```

2.
```
X <- matrix(rnorm(11*1000, mean=rep(-5:5, 1000), sd=rep(seq(0.1, 2, length.out=11)
apply(X, 1, mean)
apply(X, 1, sd)
# Note that since these are simulations, the means and standard deviations should
```

3.
```
means <- rowMeans(X)
sds <- apply(X, 1, sd)
X <- sweep(X, 1, means, "-")
X <- sweep(X, 1, sds, "/")
apply(X, 1, mean)
apply(X, 1, sd)
# Note that the means may be just very close to zero (e.g. 10^-15) but not actuall
```

4.
```
sumBelow <- function(x, a){
  return(sum(x[x<a]))
}
apply(X, 1, sumBelow, a=0)
apply(X, 1, sumBelow, a=1)
```

5.
```
par(mfrow=c(2,2))
apply(X[4:7,], 1, hist)
```

18.0.38 S3 classes

Corresponding exercises: see Section 15.5.1

1.
```
circle <- function(radius){
  if (radius < 0)
    stop("Radius must be positive!")

  c <- list(radius = radius)

  class(c) <- "circle"
  return(c)
}

smallCircle <- circle(0.01)
largeCircle <- circle(1000)
# invalidCircle <- circle(-1)
```

2.
```
print.circle <- function(x){
  output <- paste0("Here's a circle with radius = ", x$radius, ".")
  print(output)
}
```

```r
# call print
print(smallCircle)
print(largeCircle)

# Note that you should never call the S3 function itself (e.g. never call print.circle(small
```

3. ```r
   plot.circle <- function(x){
     grid.circle(r = x$radius)
   }

   # call plot
   plot(smallCircle)
   plot(largeCircle) # the circle is so large that it doesn't fit the window, so don't worry if

   # This is nice, isn't it? Now we can plot circles with plot().
   # No one needs to remember a complicated name for plotting a circle.
   ```

4. ```r
   # generic function
   perimeter <- function(x){
     UseMethod("perimeter", x)
   }

   # S3 function
   perimeter.circle <- function(x){
     return(2 * pi * x$radius)
   }

   # We need to write the generic function perimeter() because this is not a built-in
   # generic function of R.
   # On the other hand, print() and plot() are built-in generic functions, so we didn't need
   # to write them again.
   ```

5. ```r
   # generic function
   area <- function(x){
     UseMethod("area", x)
   }

   # S3 function
   area.circle <- function(x){
     return(pi * x$radius * x$radius)
   }
   ```

6. ```r
   # small circle
   perimeter(smallCircle)
   ```

```r
area(smallCircle)

# large circle
perimeter(largeCircle)
area(largeCircle)
```

7.
```r
# constructor for class rectangle
rectangle <- function(width, height){
  if (width < 0) stop("Width must be positive!")
  if (height < 0) stop("Height must be positive!")

  r <- list(width = width, height = height)

  class(r) <- "rectangle"
  return(r)
}

# create some objects class rectangle
smallRec <- rectangle(0.01, 0.05)
largeRec <- rectangle(50, 100)
# invalidRec <- rectangle(-1, 0.01)

# S3 function for print
print.rectangle <- function(x){
  output <- paste0("Here's a rectangle with width = ", x$width, " and height = ",
  print(output)
}

# call print
print(smallRec)
print(largeRec)

# S3 function for plot
plot.rectangle <- function(x){
  grid.rect(x$width, x$height)
}

# call plot
plot(smallRec)
plot(largeRec) # the rectangle is so large that it doesn't fit the window, so don

# S3 function for perimeter (Note: we don't need to define the perimeter() generic
perimeter.rectangle <- function(x){
  return(2 * (x$width + x$height))
}
```

```r
# S3 function for area (Note: we don't need to define the area() generic function again!)
area.rectangle <- function(x){
  return(x$width * x$height)
}

# check if perimeter() and print() are working!
perimeter(smallRec)
area(smallRec)

perimeter(largeRec)
area(largeRec)
```

8. `print()` is a generic function and will call `UseMethod("print", x)`. `UseMethod()` will then search for a print method for class "mammal". However, we haven't defined such a method. It will therefore call the `print.default()` method, which will do some default printing.

9. ```r
   print.mammal <- function(x, ...){
     print(paste0("This is ", x$name, ". It's a ", x$color, " ", x$sex, " mammal."))
   }

   print(bobby)
   # print() is a generic function and calls UseMethod("print", x).
   # UseMethod() searches for a print method for class "mammal".
   # As we've named our function print.mammal() - which is the correct syntax for S3 function -
   # R will call print.mammal()!
   ```

10. ```r
    class(bobby) <- c(class(bobby), "cow")

    print(bobby)
    # still prints the function print.mammal()
    # Reason: class "mammal" comes before class "cow" in the vector of class attributes.
    ```

11. ```r
    class(bobby) <- rev(class(bobby))

    print(bobby)
    # still prints the function print.mammal().
    # Although class "cow" comes before class "mammal" in the vector of class attributes,
    # class "cow" does not have a printing function yet.
    # Therefore, R goes to the second element in the vector - "mammal" - and calls the correspon
    ```

12. ```r
    print.cow <- function(x, ...){
        print(paste0("This is ", x$name, ". It's a ", x$color, " ", x$sex, " cow."))
    }
    ```

```
print(bobby)
# As we've now defined a print function for class "cow", R calls this one!
# Make sure that "cow" comes before "mammal" in the vector of classes.
```

18.0.39 Tidyverse

Corresponding exercises: see Section 16.1.9

1.
```
data("iris")
filter(iris, Sepal.Length %in% c(5.0, 6.0, 7.0))
```

2.
```
data("ChickWeight")
ChickWeight %>%
  filter(Diet != 4) %>%
  group_by(Time) %>%
  summarise(mean_weight = mean(weight))
```

18.0.40 ggplot2

Corresponding exercises: see Section 17.0.2

1.
```
library(datasets)
data("CO2")

ggplot(CO2,aes(x=conc, y=uptake))+
  geom_point(aes(color=Type))+
  ggtitle("CO2 uptake in plants") + xlab("carbon dioxide concentrations (mL/L)") +
  theme_bw()
```

2.
```
library(datasets)
data("CO2")

ggplot(CO2,aes(x=conc, y=uptake))+
  geom_point(aes(color=Type))+
  ggtitle("CO2 uptake in plants") + xlab("carbon dioxide concentrations (mL/L)") +
  facet_wrap(~Treatment)+
  geom_smooth(aes(group=Treatment))+
  theme_bw()
```