Data Analysis & Visualization using R (1)

Michiel Noback

2020-03-21

Contents

Chapter 1

Getting started

Welcome, you have landed at the eBook accompanying my R course for Life Science students, ***Data Analysis and Visualization using R (DAVuR)***.

Before reading on, you should check whether you are ready to work with R on your own computer. You should have installed R, RStudio and Tinytech or some other Latex alternative for your OS.

This eBook is the result of many hours of work and has been finetuned after lecturing the material for some years. You are free to use it in any way you like: courses and self-paced study.

Chapter 2

The toolbox

2.1 Why do statistical programming?

Since you're a life science student -that is my target audience at least-, you have probably worked with Excel or SPSS at some time. Have you ever wondered

- Why am I doing this exact same series of mouse clicks again and again? Is there not a more efficient way?
- How can I describe my work reproducibly as a series of mouse clicks?

If so, then R may be your next favourite data analysis tool. It takes a little effort at first, but once you get the hang of it you will never create a plot in Excel again.

With R - as with any programming language,

- Redoing an analysis or generating a report with minor adjustments is a breeze
- The analysis is central, not the output. This guarantees complete reproducibility

2.2 Overview of the toolbox

This chapter will introduce you to a toolbox that will serve you well during your data quests.
It consists of

- The R programming language and builtin functionality
- The RStudio Integrated Development Environment (IDE)
- R Markdown as documenting and reporting tool

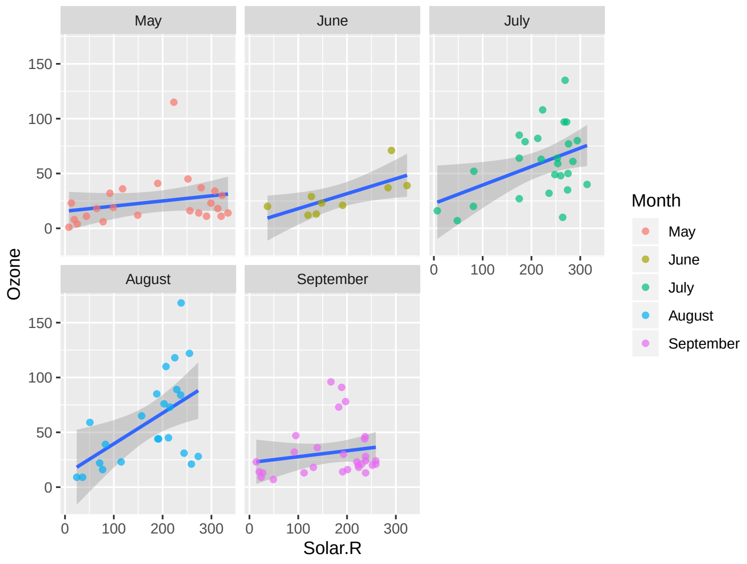


Figure 2.1: A facetplot - multiple similar plots split over a single nominal or ordinal variable

2.2.1 Tool 1: The R programming language

Nobody likes to pay for computer tools. R is completely free of charge. Moreover, it is completely open source. This is of course one of the main reasons for its popularity; other statistical tools are not free and sometimes downright expensive. Besides this free nature, R is very popular because it has an interactive mode. We call this a read–evaluate–print loop: REPL. This means you don't need to write programs to run code. You simply type a command in the ***console***, press ennter and immediately get the result on the line below.
As stated above, because you store your analyses in code, repeating these analyses -possibly with with new data or changed settings- is very easy. One of my personal favorite features is that R supports "literate programming" for creating presentations (such as this one!) and other publications (reports, papers etc). Pdf documents, Microsoft Word documents, web pages (html) and e-books are all possible outputs of a single RMarkdown master document.

Finally, R has advanced embedded graphical support. This means that graphical output (a plot) is as easy to generate as textual output!

Here are some figures to whet your appetite. You will be able to create all of these yourself at the end of this course (actually, a pair of courses).

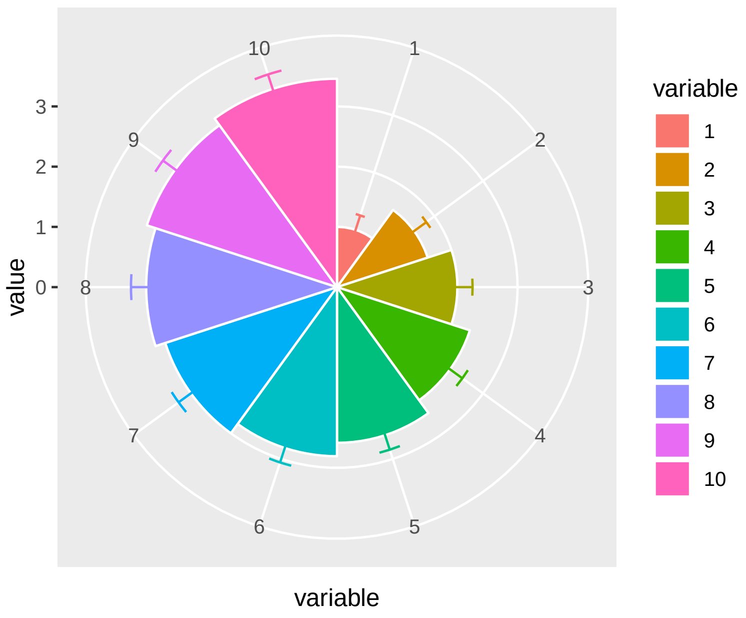


Figure 2.2: A polar plot - the dimensions are not your normal 2d x and y

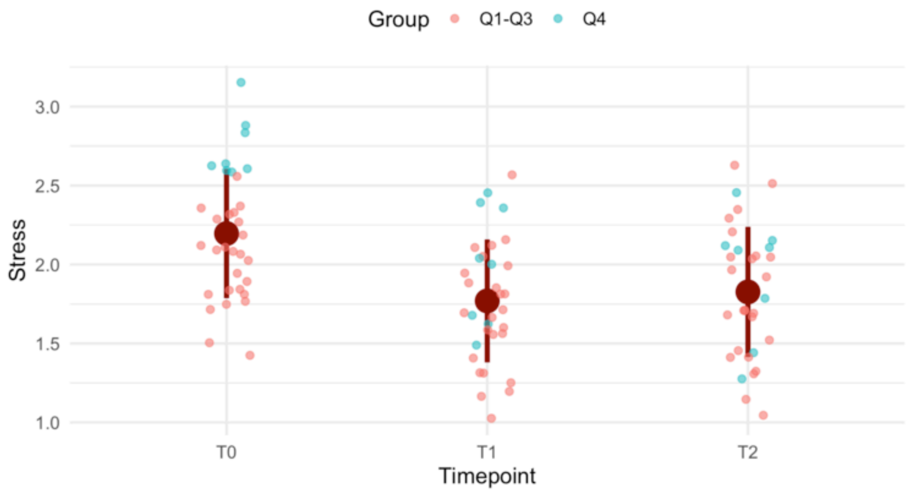
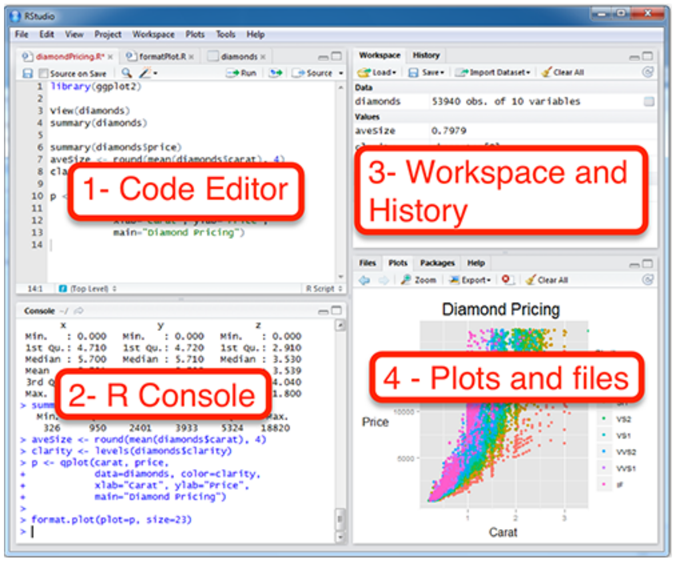


Figure 2.3: A custom jitter visualization

Figure 2.4: RStudio logo

2.2.2 Tool 2: RStudio as development environment

RStudio is a so-called Integrated Development Environment. This means it is a “Swiss Mulitool” for programming. With it, you manage and run code, files, documentation on the language (help pages), building different output formats. The workbench has several panels and looks like this when you run the application.



```
library(ggplot2)

View(diamonds)
summary(diamonds)

summary(diamonds$price)
aveSize <- round(mean(diamonds$carat), 4)
cla

p <


             main="Diamond Pricing")
```

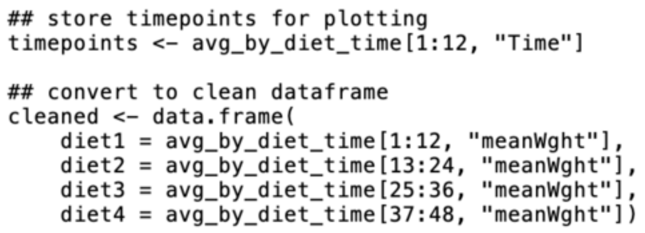
```
Data
diamonds      53940 obs. of 10 variables
Values
aveSize       0.7979
```

```
        x                 y                 z
Min.   : 0.000   Min.   : 0.000   Min.   : 0.000
1st Qu.: 4.710   1st Qu.: 4.720   1st Qu.: 2.910
Median : 5.700   Median : 5.710   Median : 3.530
Mean                                       3.539
3rd Q                                      4.040
Max.                                       1.800
> summ
   Min.                                  Max.
    326     950    2401    3933    5324   18820
> aveSize <- round(mean(diamonds$carat), 4)
> clarity <- levels(diamonds$clarity)
> p <- qplot(carat, price,
+            data=diamonds, color=clarity,
+            xlab="Carat", ylab="Price",
+            main="Diamond Pricing")
>
> format.plot(plot=p, size=23)
> |
```

You primarily work with 4 panels of the workbench:

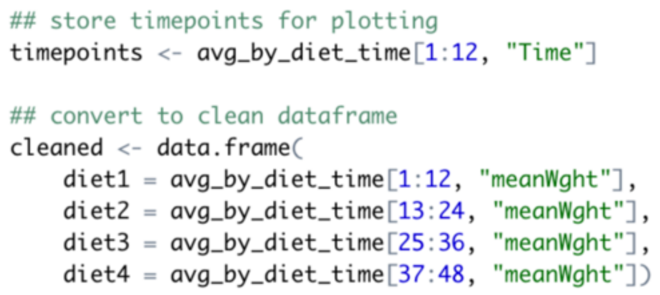
1. **Code editor** where you write your scripts and RMarkdown documents: text files with code you want to execute more than once
2. **R console** where you execute lines of code one by one
3. **Environment and History** See what data you have in memory, and what you have done so far
4. **Plots, Help & Files**

You use the console to do basic calculations, try pieces of code, develop a function, or load scripts (from the code editor) into memory. On the other hand,



```
## store timepoints for plotting
timepoints <- avg_by_diet_time[1:12, "Time"]

## convert to clean dataframe
cleaned <- data.frame(
    diet1 = avg_by_diet_time[1:12, "meanWght"],
    diet2 = avg_by_diet_time[13:24, "meanWght"],
    diet3 = avg_by_diet_time[25:36, "meanWght"],
    diet4 = avg_by_diet_time[37:48, "meanWght"])
```

Figure 2.5: code in TextEdit



```
## store timepoints for plotting
timepoints <- avg_by_diet_time[1:12, "Time"]

## convert to clean dataframe
cleaned <- data.frame(
    diet1 = avg_by_diet_time[1:12, "meanWght"],
    diet2 = avg_by_diet_time[13:24, "meanWght"],
    diet3 = avg_by_diet_time[25:36, "meanWght"],
    diet4 = avg_by_diet_time[37:48, "meanWght"])
```

Figure 2.6: exact same file in RStudio editor

the code editor is used to work on code that has life span longer than a few minutes: analyses you may want to repeat, or develop further in the form of scripts and RMarkdown documents. The code editor supports many file types for viewing and editing: regular text, structured datafiles (text, csv, data files), scripts (programs), and analytical notebooks (RMarkdown).

What is nice about the ***code editor*** above regular text editors such as Notepad, Wordpad, TextEdit, is that it knows about different file types and their constituting elements and helps your read, write (autocomplete, error alerts), scan and organize them by displaying these elements using coloring, font types and other visual aids.

Here is the same piece of code, which is a plain text file, in two different editors. First as plain text in the Mac TextEdit app and next in the RStudio code editor:

It is clearly visible where the code elements, numeric data and character data are within the code.

2.2.3 Tool 3: RMarkdown

In RMarkdown, you can combine regular text and figures with embedded R code that will be executed to generate a final document.

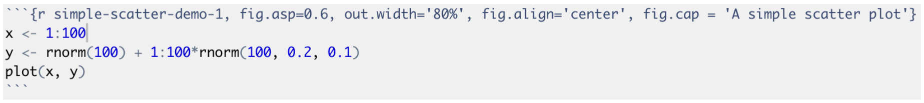
You can use it to create reports in word, pdf or web (html); create presentations (pdf or web); create entire ebooks and websites (such as this one). This entire ebook itself is written in RMarkdown!

Markdown is a very basic ***markup*** language. Markup means that you use textual elements to indicate structure instead of content. RMarkdown simply is Markdown with embedded pieces of R code. Consider this piece of Markdown:

```
### Tool 3: RMarkdown

![](figures/markdown_logo.jpg)

In RMarkdown, you can combine regular text and figures with embedded R code that will
```

The result of this snippet is the top of the current paragraph you are reading.

Here is a piece of R code we call a ***code chunk*** that plots some random data in a scatter plot. In RStudio this piece of R code within (the current) RMarkdown document looks like this:



~~~
```{r simple-scatter-demo-1, fig.asp=0.6, out.width='80%', fig.align='center', fig.cap = 'A simple scatter plot'}
x <- 1:100
y <- rnorm(100) + 1:100*rnorm(100, 0.2, 0.1)
plot(x, y)
```
~~~

Next, when I ***knit*** the document into web format it results in the piece below together with its output, a scatter plot.

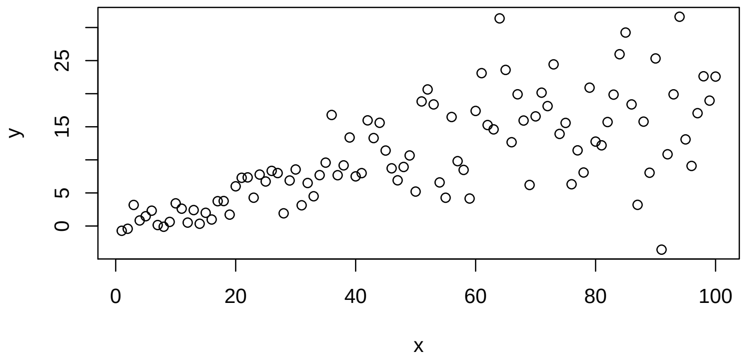


Figure 2.7: A simple scatter plot

```
x <- 1:100
y <- rnorm(100) + 1:100*rnorm(100, 0.2, 0.1)
plot(x, y)
```

RMarkdown is really basic; in fact it is translated into html, the markup language of the web, before any further processing occurs. That is why you can also embed html elements within it. Here are the most basic elements you can use in Markdown documents.

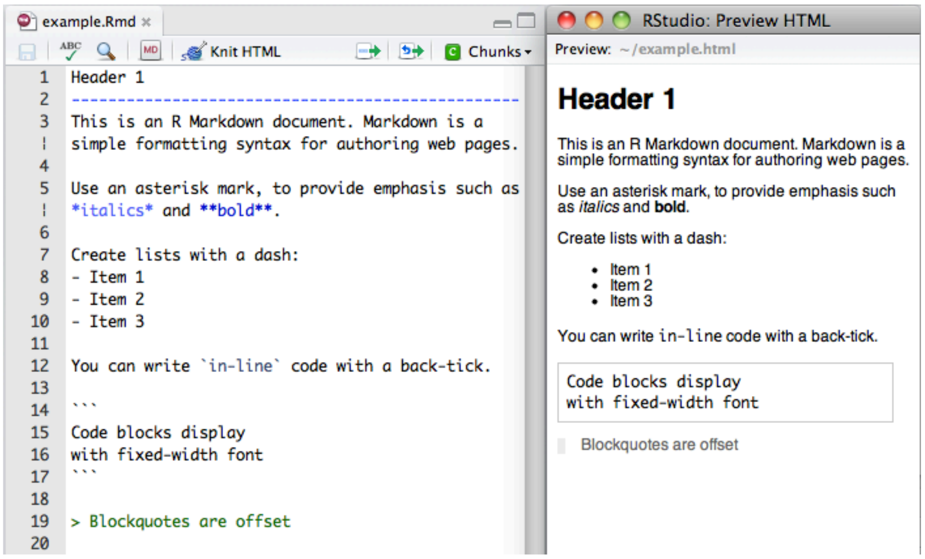
Finally, it is also possible to embed Latex elements. For instance, equations can be defined in a text format. This:

`$$d(p, q) = \sqrt{\sum_{i = 1}^{n}(q_i-p_i)^2}$$`

results in this:

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}$$

Happy coding!



```
Header 1
---------------------------------------------------
This is an R Markdown document. Markdown is a
simple formatting syntax for authoring web pages.

Use an asterisk mark, to provide emphasis such as
*italics* and **bold**.

Create lists with a dash:
- Item 1
- Item 2
- Item 3

You can write `in-line` code with a back-tick.

```
Code blocks display
with fixed-width font
```

> Blockquotes are offset

```

# Header 1

This is an R Markdown document. Markdown is a simple formatting syntax for authoring web pages.

Use an asterisk mark, to provide emphasis such as *italics* and **bold**.

Create lists with a dash:

- Item 1
- Item 2
- Item 3

You can write `in-line` code with a back-tick.

```
Code blocks display
with fixed-width font
```

Blockquotes are offset

Figure 2.8: RMarkdown

Chapter 3

Basic R - coding basics

3.1 First look at vectors, fuctions and variables

3.1.1 Doing Math in the console

The console is the place where you do quick calculations, tests and analyses that do not need to be saved (yet) or repeated. It is the the tab that says "Console" and on first use, R puts it in the lower left panel.

In the console, the ***prompt*** is the "greater than" symbol ">". R waits here for you to enter commands. When the panel has "focus" the cursor is blinking on and off. You can use the console as a calculator. It supports all regular math operations, in the way you would expect them:

`+` : 'plus', as in 2 + 2 = 4

`-` : 'subtract', as in 2 - 2 = 0

`*` : 'multiply', as in 2 * 3 = 6

`/` : 'divide', as in 8 / 4 = 2

`^` : 'exponent', as in $2^3 = 8$. In R, `^` is synonym of `**`

For the square root you can use $n^{0.5}$: `n**0.5`, or the function `sqrt()` (discussed later).

When Enter is pressed when the mathematical statement is not complete yet, the `>` symbol is replaced by a `+` at the start of the new line, indicating the statement is a continuation. Here is an example:

```
> 1 + 3 + 4 +
+
```

So the `+` at the start of line 2 is not a mathematical `+` but a “continuation symbol”. You can always abort the current statement by pressing Escape.

When a statement *is* complete, the result will be printed in the next line:

```
> 31 + 11
[1] 42
```

The result is of course `42`; the leading `[1]` is the ***index*** of the result. We will address this later.

Operator Precedence

All “operators” adhere to the standard mathematical **precedence** rules (PEMDAS):

```
    Parentheses (simplify inside these)
    Exponents
    Multiplication and Division (from left to right)
    Addition and Subtraction (from left to right)
```

With complex statements you should be aware of operator precedence! If you are not sure, or want to make your expression less ambiguous you should simply use parentheses `()` because they have highest precedence.

Besides math operators, R knows a whole set of other operators. They will be dealt with later in this chapter.

> ***Programming Rule*** Always place spaces around both sides of an operator, with the exception of `^` and `**`.

3.1.2 An expression dissected

When you type `21 / 3` this called an ***expression***. The expression has three parts: an operator (`/` in the middle) and two operands. The left operand is `21` and the right operand is `3`.
Since there is no assignment, the result of this expression will be send to the console as output, giving `[1] 7`.

Because this expression is the sole contents of the current line in the console, it is also called a ***statement***.

> ***Statement vs expression*** A statement is a complete line of code that performs some action, while an expression is any section of code that evaluates to a value.

Ending statements

In R, the newline (enter) is an end-of-statement character. Optionally you can end statements with a semicolon “;”. However, when you have more statements on a single line they are mandatory is in this example:

```r
x <- c(1, 2, 3); x; x <- 42; x
```

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

```
## [1] 42
```

> ***Programming Rule***: Have one statement per line and don't use semicolons

Comments

Everything on a line after a hash sign "`#`" will be ignored by R. Use it to add explanation to your code:

```r
## starting cool analysis
x <- c(T, F, T) # Creating a logical vector
y <- c(TRUE, FALSE, TRUE) # same
```

3.2 Functions

Simple mathematics is not the core business of R.

Going further than basic math, you will need functions, mostly pre-existing functions but often also custom functions that you write yourself. Here is a definition of a function:

> *A function is a piece of functionality that you can execute by typing its name, followed by a pair of parentheses. Within these parentheses, you can pass data for the function to work on. Functions often, but not always, return a value.*

Function usage -or a ***function call***- has this general form:

$$function\_name(arg_1, arg_2, ..., arg_n)$$

Example: Square root with `sqrt()`

You have already seen that the square root can be calculated as $n^{0.5}$. However, there is also a function for it: `sqrt()`. It ***returns*** the square root of the given ***parameter***, a number, *e.g.* `sqrt(36)`

```r
36^0.5
sqrt(36)
```

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

Another example: `paste()`

The `paste()` function can take any number of arguments and returns them, combined into a single text (character) string. You can also specify a separator using `sep="<separator string>"`:

```r
paste(1, 2, 3, sep = "---")
```

```
## [1] "1---2---3"
```

Note the use of quotes surrounding the dashes: `"---"`; they indicate it is text, or character, data.
Also note the use of a name for only the last argument. Not all arguments can be specified by name, but when possible this has preference, as in `sep = "---"`.

3.2.1 Getting help on a function

Type `?function_name` or `help(function_name)` in the console to get help on a function. The function documentation will appear in the panel containing the `Help` tab, Its location is dependent on your set of preferences.
For instance, typing `?sqrt` will give the help page of the square root function together with the `abs()` function.
R help pages always have the exact same structure:

- Name & package (e.g. `{base}`)
- Short description
- Description
- Usage
- Arguments
- Details
- …
- Examples

Scroll down in the help to see example usages of the function. Alternatively, type `example(sqrt)` in the console to have all examples executed in order, until you press Escape.

3.3 Variables

In math and programming you often use variables to label or name pieces of data, or a function in order to have them reusable, retrievable, changeable.

> A ***variable*** is a named piece of data stored in memory that can be accessed via its name

For instance, `x = 42` is used to define a variable called `x`, with a value attached to it of `42`. Variables are really *variable* - their value can change! In R you usually assign a value to a variable using "`<-`", so "`x <- 42`" is equivalent to "`x = 42`". Both will work in R, but the "arrow" notation is preferred.

3.4 Vectors

3.4.1 R is completely vector-based

In R, ***all data lives inside vectors***. When you type '2 + 4', R will execute the following series of actions:

1. create a vector of length 1 with its element having the value 2
2. create a vector of length 1 with its element having the value 4
3. add the value of the second vector to ALL the values of vector one, and recycle any shorter vector as many times as needed

Step 3 is a crucial one. It is essential to grasp this aspect in order to understand R. Therefore we'll revisit it later in more detail.

3.4.2 Five datatype that live in vectors

R knows five basic types of data:

type	descripton	examples
numeric	numbers with a decimal part	'3.123', '5000.0', '4.1E3'
integer	numbers without a decimal part	'1', '0', '2999'
logical	Boolean values: yes/no)	'true' 'false'
character	text, should be put within quotes	''hello R'' '"A cat!"'
factor	nominal and ordinal scales	\<dealt with later\>

All these types are created in similar ways, and can often be converted into other types.

Note 1: If you type a number in the console, it will always be a `numeric` value, decimal part or not.
Note 2: For character data, single and double quotes are equivalent but double are preferred; type `?Quotes` in the console to read more on this topic.

3.4.3 Creating vectors

You will see shortly that there are many ways to create vectors: a custom collection, a series, a repetition of a smaller set, a random sample from a distribution, etc. etc.

The simplest way to create a vector is the first: create a vector from a custom set of elements, using the "Concatenate" function `c()`. The `c()` function simply takes all its arguments and puts them behind each other, in the order in which they were passed to it, and returns the resulting vector.

```
> c(2, 4, 3)
```

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

```
> c("a", "b", c("c", "d"))
```

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

```
> c(0.1, 0.01, 0.001)
```

```
## [1] 0.100 0.010 0.001
```

```
> c(T, F, TRUE, FALSE) # There are two way to write logical values
```

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

Vectors can hold only one data type

A vector can hold only one type of data. Therefore, if you pass a mixed set of values to the function `c()`, it will **coerce** all data into one type. The preferred type is numeric. However, when that is not possible the result will most often be a character vector. In the example below, two numbers and a character value are passed. Since `"a"` cannot be coerced into a numeric, the returned vector will be a character vector.

```
c(2, 4, "a")
```

```
## [1] "2" "4" "a"
```

Here are some more coercion examples.

```
> c(1, 2, TRUE) # To numeric
```

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

```
> c(TRUE, FALSE, "TRUE") # To character
```

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

```
> c(1.3, TRUE, "1") # To character
```

```
## [1] "1.3"  "TRUE" "1"
```

Using the function `class()`, you can get the data type of any value or variable.

```
> class(c(2, 4, "a"))
```

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

```
> class(1:5)
```

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

```
> class(c(2, 4, 0.3))
```

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

```
> class(c(2, 4, 3))
```

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

3.4.4 Vector fiddling

Vector arithmetic

Let's have a look at what it means to work with vectors, as opposed to singular values (also called *scalars*). An example is probably best to get an idea.

```
x <- c(2, 4, 3, 5)
y <- c(6, 2)
x + y
```

```
## [1] 8 6 9 7
```

As you can see, R works ***set based*** and will ***cycle*** the shorter of the two operands to deal with all elements of the longer operand. How about when the longer one is not a multiple of the shorter one?

```
x <- c(2, 4, 3, 5)
z <- c(1, 2, 3)
x - z
```

```
## Warning in x - z: longer object length is not a multiple of shorter object
## length
```

```
## [1] 1 2 0 4
```

As you can see this generates a warning that "longer object length is not a multiple of shorter object length". However, R will proceed anyway, cycling the shorter one.

3.5 Other operators

Here is a complete listing of operators in R. Some operators such as `^` are *unary*, which means they have a single *operand*; a single value or they operate on. On the other hand, *binary* operators such as `+` have two *operands*.

The following unary and binary operators are listed in precedence groups, from highest to lowest. Many of them are still unknown to you of course. We will encounter most of these along the way as the course progresses, starting with a few in this section.

operator	purpose
:: :::	access variables in a namespace
$ @	component / slot extraction
[[[	indexing
^	exponentiation (right to left)
- +	unary minus and plus
:	sequence operator
%any%	special operators (including %% and %/%)
* /	multiply, divide
+ -	(binary) add, subtract
< > <= >= == !=	ordering and comparison
!	negation
& &&	and
| ||	or
~	as in formulae
-> -»	rightwards assignment
<- «-	assignment (right to left)
=	assignment (right to left)
?	help (unary and binary)

3.5.1 Logical operators

Logical operators are used to evaluate and/or combine expressions that result in a single logical value: `TRUE` or `FALSE`. The ***comparison operators*** compare two values (numeric, character - any type is possible) to get to a logical value, but always set-based! In the following chunk, each of the values in `x` is considered and if it is smaller than or equal to the value `4`, `TRUE` is returned, else `FALSE`.

```
x <- c(1, 5, 4, 3)
x <= 4
```

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

Other comparison operators are `<` (less then), `<=` (less then or equal to), `>` (greater then), `>=` (greater then or equal to), and `==` (equal to).

Another category of logical operators is the set of ***boolean operators***. These are used to *reduce* two logical values into one. These are

- `&`: logical "AND"; `a & b` will evaluate to `TRUE` only if `a` AND `b` are TRUE.

- `|`: logical "OR"; `a | b` will evaluate to `TRUE` only if `a` OR `b` are TRUE, no matter which.
- `!`: logical -unary- "NOT"; negates the right operand: `! a` will evaluate to the "flipped" logical value of `a`.

Here is a more elaborate example combining comparison and boolean operators. Suppose you have vectors a and b and you want to know which values in `a` are greater than in `b` and also smaller than `3`. This is the expression used for answering that question.

```
a <- c(2, 1, 3, 1, 5, 1)
b <- c(1, 2, 4, 2, 3, 0)
a > b & a < 3 ## returns a logical vector with test results
```

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

Here is a special case. Can you figure out what happens there?

```
6 - 2 : 5 < 3
```

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

Calculations with logical vectors

Quite often you want to know how many cases fit some condition. A convenient thing in that case is that logical values have a numeric counterpart or "hidden face":

```
- TRUE == 1
- FALSE == 0
```

- Use `sum()` to use this feature

```
x <- c(2, 4, 2, 1, 5, 3, 6)
x > 3 ## which values are greater than 3?
```

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

```
sum(x > 3) ## how many are greater than 3?
```

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

3.5.2 Modulo: %%

The modulo operator gives the remainder of a division.

```
10 %% 3
```

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

```
4 %% 2
```

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

```
11 %% 3
```

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

The modulo is most often used to establish periodicity: `x %% 2` is zero for all even numbers. Likewise, `x %% 10` will be zero for every tenth value.

3.5.3 Integer division %/% and rounding

The integer division is the complement of modulo and gives the integer part of a division, it simply "chops off" the decimal part.

```
10 %/% 3
```

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

```
4 %/% 2
```

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

```
11 %/% 3
```

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

Note that `floor()` does the same. In the same manner, `ceiling()` rounds up to the nearest integer, no matter how large the decimal part. Finally, there is the `round()` method to be used for - well, rounding. Be aware that rounding in R is not the same as rounding your course grade which always goes up at `x.5`. Rounding `x.5` values mathematically goes to the nearest even number:

```
x <- c(0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5)
round(x, 0)
```

```
## [1] 0 2 2 4 4 6 6 8
```

3.5.4 The %in% operator

The `%in%` operator is very handy when you want to know if the elements of one vector are present in another vector. An example explains best, as usual:

```
a <- c("one", "two", "three")
b <- c("zero", "three", "five", "two")
a %in% b
b %in% a
```

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

There is no positional evaluation, it simply reports if the corresponding element in the first is present *anywhere* in the second.

3.6 Vector creation methods

Since vectors are the bricks with which *everything* is built in R, there are many, many ways to create them. Here, I will review the most important ones.

Method 1: Constructor functions

Often you want to be specific about what you create: use the class-specific constructor **OR** one of the conversion methods. Constructor methods have the name of the type. They will create and return a vector of that type wit as length the number that is passed as constructor argument:

```
> integer(4)
```

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

```
> character(4)
```

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

```
> logical(4)
```

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

Method 2: Conversion functions

Conversion methods have the name `as.XXX()` where XXX is the desired type. They will attempt to coerce the given input vector into the requested type.

```
x <- c(1, 0, 2, 2.3)
class(x)
```

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

```
as.logical(x)
```

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

```
as.integer(x)
```

```
## [1] 1 0 2 2
```

But there are limits to coercion: R will not coerce elements with types that are non-coercable: you get an `NA` value.

```
x <- c(2, 3, "a")
y <- as.integer(x)
```

```
## Warning: NAs introduced by coercion
```

```
class(y)
```

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

```
y
```

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

Method 3: The colon operator

The colon operator *(:)* generates a series of integers fromthe left operand to -and including- the right operand.

```
1 : 5
```

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

```
5 : 1
```

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

```
2 : 3.66
```

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

Method 4: The `rep()` function

The `rep()` function takes three arguments. The first is an input vector. The second, `times =`, specifies how often the *entire* input vector should be repeated. The second argument, `each =`, specifies how often *each* individual element from the input vector should be repeated. When both arguments are provided, `each =` is evaluated first, followed by `times =`.

```
rep(1 : 3, times = 3)
```

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

```
rep(1 : 3, each= 3)
```

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

```
rep(1 : 3, times = 2, each = 3)
```

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

Method 5: The `seq()` function

The `seq()` function is used to create a numeric vector in which the subsequent element show sequential increment or decrement. You specify a range and a step which may be neative if the range end (`to =`) is lower than the range start (`from =`).

```
> seq(from = 1, to = 3, by = .2)
```

```
##  [1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0
```

```
> seq(1, 2, 0.2) # same
```

```
## [1] 1.0 1.2 1.4 1.6 1.8 2.0
```

```
> seq(1, 0, length.out = 5)
```

```
## [1] 1.00 0.75 0.50 0.25 0.00
```

```
> seq(3, 0, by = -1)
```

```
## [1] 3 2 1 0
```

Method 6: Through vector operations

Of course, new vectors, often of different type, are created when two vectors are combined in some operation, or a single vector is processed in some way.

This operation of two numeric vectors results in a logical vector:

```
1:5 < c(2, 3, 2, 1, 4)
```

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

And this `paste()` call results in a character vector:

```
paste(0:4, 5:9, sep = "-")
```

```
## [1] "0-5" "1-6" "2-7" "3-8" "4-9"
```

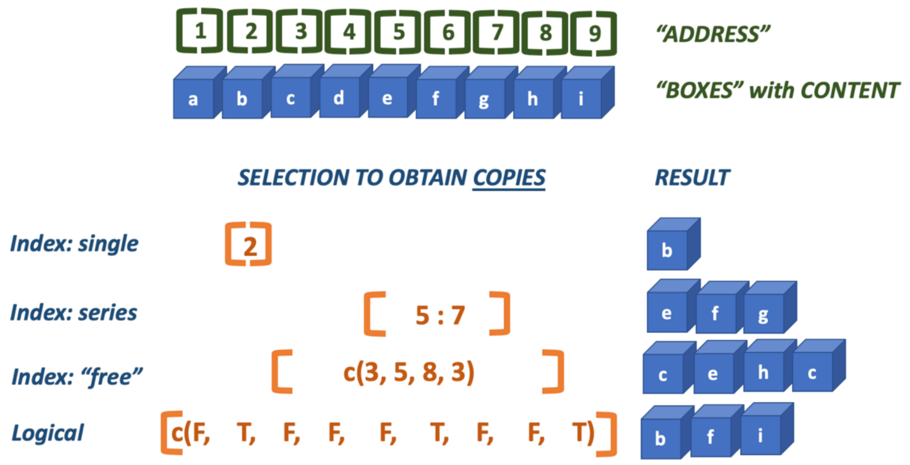
3.7 Selecting vector elements

You often want to get to know things about specific values within a vector

- what value is at the third position?
- what is the highest value?
- which positions have negative values?
- what are the last 5 values?

There are two principal ways to do this: through indexing with positionional reference ("addresses") and through logical indexing.

Here is a picture that demonstrates both.



The `index` is the position of a value within a vector. R starts at one (1), and therefore ends at the length of the vector. Brackets `[]` are used to specify one or more indices that should be selected (returned).

Here are two examples of straightforward indexing, selecing a single or a series of elements.

```
x <- c(2, 4, 6, 3, 5, 1)
x[4] ## fourth element
```

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

```
x[3:5] ## elements 3 to 5
```

```
## [1] 6 3 5
```

However, the technique is much more versatile. You can use indexing to select elements multiple times and thus create copies of them, or select elements in any order you desire.

```
x[c(1, 2, 2, 5)] ## elements 1, 2, 2 and 5
```

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

```
x <- c(2, 4, 6, 3, 5, 1)
```

Besides integers you can use logicals to perform selections:

```
x[c(T, F, T, T, T, F)]
```

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

As with all vector operations, shorter vectors are cycled as often as needed to cover the longer one:

```
x[c(F, T, F)]
```

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

In practice you won't type literal logicals very often; they are ususaly the result of some comparison operation. Here, all even numbers are selected because their modulo will retun zero.

```
x[x %% 2 == 0]
```

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

And all of the maximum values in a vector are retreived:

```
x <- c(2, 3, 3, 2, 1, 3)
x[x == max(x)]
```

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

There is a caveat in selecting the last n values: the colon operator has highest precedence! Here, the last two elements are (supposed to be selected).

```
x <- c(2, 4, 6, 3, 5, 1)
x[length(x) - 1 : length(x)] #fails
```

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

```
x[(length(x) - 1) : length(x)] ## parentheses required!
```

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

Use `which()` to get an index instead of value

The function `which()` returns indices for which the logical test evaluates to `true`:

```
which(x >= 2) ## which positions have values 2 or greater?
```

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

```
which(x == max(x)) ## which positions have the maximum value?
```

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

3.8 Some coding style rules rules for writing code

- Names of variables start with a lower-case letter
- Words are separated using underscores
- Be descriptive with names
- Function names are verbs

- Write all code and comments in English
- Preferentially use one statement per line
- Use spaces on both sides of ALL operators
- Use a space after a comma
- Indent code blocks -with {}- with 4 or 2 spaces, but be consistent

Follow Hadleys' style guide http://adv-r.had.co.nz/Style.html

3.9 The best keyboard shortcuts for RStudio

- `ctr + 1` go to code editor
- `ctr + 2` go to console
- `ctr + alt + i` insert code chunk (RMarkdown)
- `ctr + enter` run current line
- `ctr + shift + k` knit current document
- `ctr + alt + c` run current code chunk
- `ctr + shift + o` source the current document

Chapter 4

Basic R - plotting basics

4.1 Basic embedded plot types

Looking at numbers is boring - people want to see pictures! Doing analyses without visualizations is like only listening to a movie.

There are a few plot types supported by base R that deal with (combinations of) vectors:

- scatter (or line-) plot
- barplot
- histogram
- boxplot

We'll only look at the bare basics in this chapter because we are going to do it for real with package `ggplot2` in the next course.

4.1.1 Scatter and line plots

Meet `plot()` - the workhorse of R plotting.

```
time <- c(1, 2, 3, 4, 5, 6)
response <- c(0.09, 0.30, 0.41, 0.48, 0.72, 1.12)
plot(x = time, y = response)
```

The function plot is used here to generate a *scatter plot*. It may generae other types of figures, depending on its input as we'll see later.

Formula notation

Instead of passing an `x =` and `y =` set of arguments, it is also possible to call the plot fuction with a ***formula notation***:

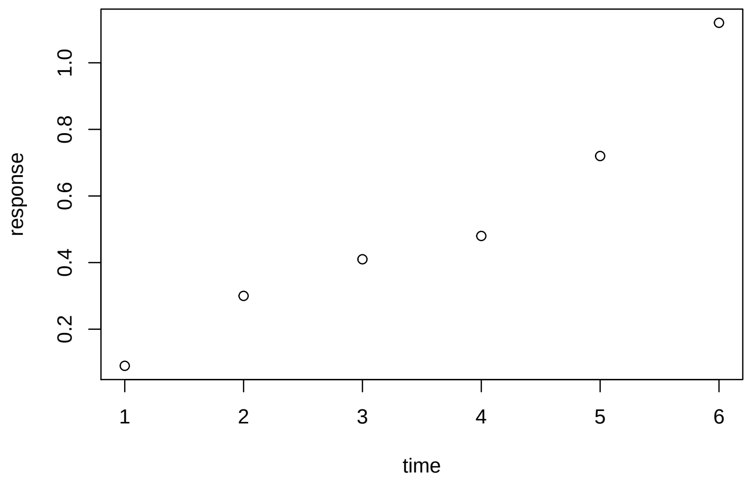
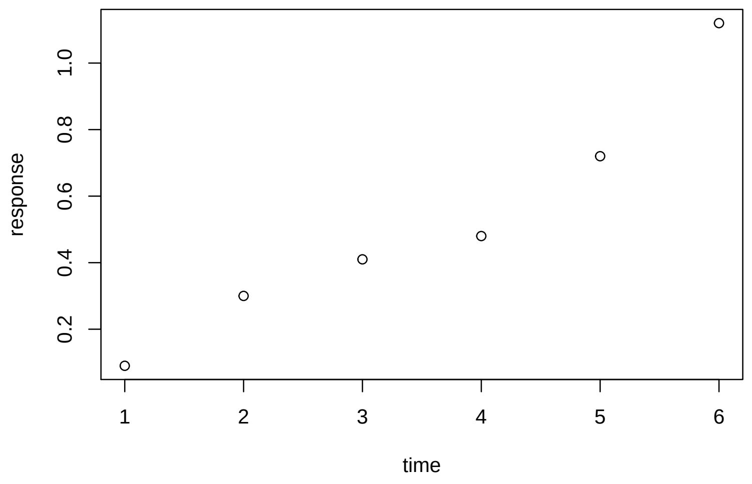


Figure 4.1: Here is a nice figure!

```r
plot(response ~ time)
```



You can read `response ~ time` as *response as a function of time*. This is a nice, readable alternative in this case, but for many functions it is the only or preferred way to specify the relationship you want to investigate.

Plot decorations

Plots should always have these decorations:

- Axis labels indicating measurement type (quantity) and its units. E.g. '[Mg] (mq/ml)' or 'Heartrate (bpm)'.
- If multiple data series are plotted: a legend

- Either a title or a figure caption, depending on the context.

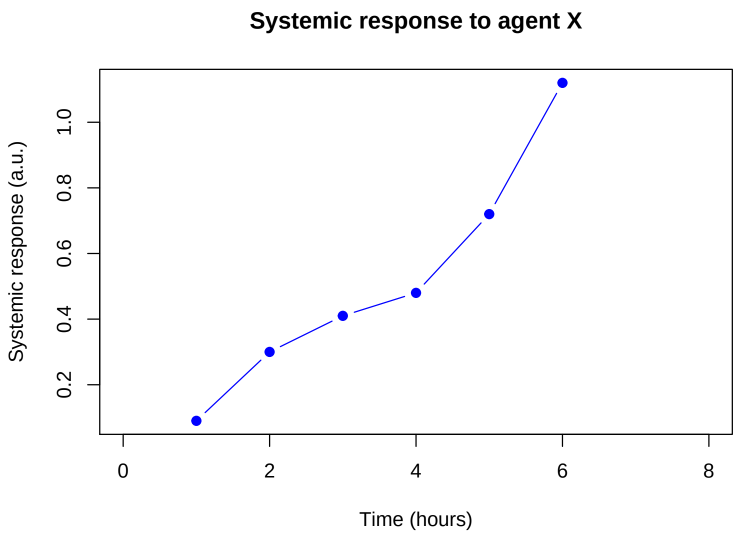
The first plots of this chapters were very bare (and a bit boring to look at): the plot has no axis labels (quantity and units) and no decoration whatsoever. By passing arguments to `plot()` you can modify or add many features of your plot. Basic decoration includes

- adjusting markers (`pch = 19`, `col = "blue"`)
- adding connector lines (`type = "b"`) or removing points (`type = "l"`)
- adding axis labels and title (`xlab = "Time (hours)"`, `ylab = "Systemic response"`, `main = "Systemic response to agent X"`)
- adjusting axis limits (`xlim = c(0, 8)`)

This is not an exhaustive listing; these are listed in the last section of this chapter.

Here is a more complete plot using a variety of arguments.

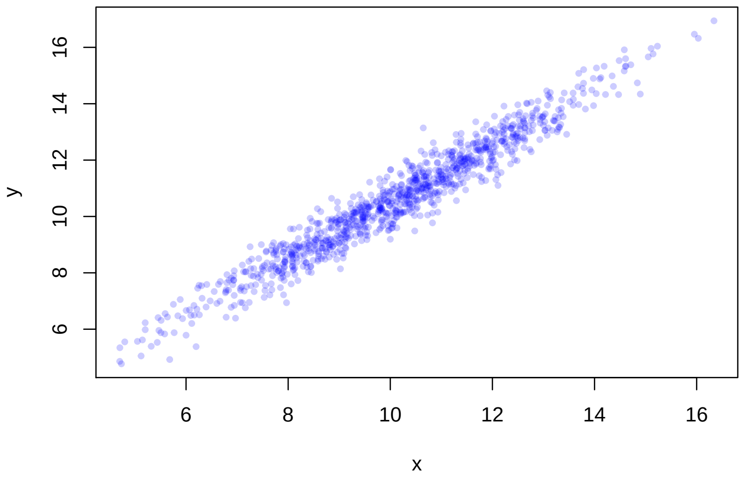
```
plot(x = time, y = response, pch = 19, type = "b", xlim = c(0, 8),
     xlab = "Time (hours)", ylab = "Systemic response (a.u.)",
     main = "Systemic response to agent X", col = "blue")
```



Adjusting the plot symbol

When you have many data points they will overlap. Using transparency with the `rgb(,, alpha=)` color definition and/or smaller plot symbols (`cex=`) solves this.

```
x <- rnorm(1000, 10, 2); y <- x + rnorm(1000, 0.5, 0.5)
plot(x, y, pch = 19, cex = 0.6,
     col = rgb(red = 0, green = 0, blue = 1, alpha = 0.2))
```



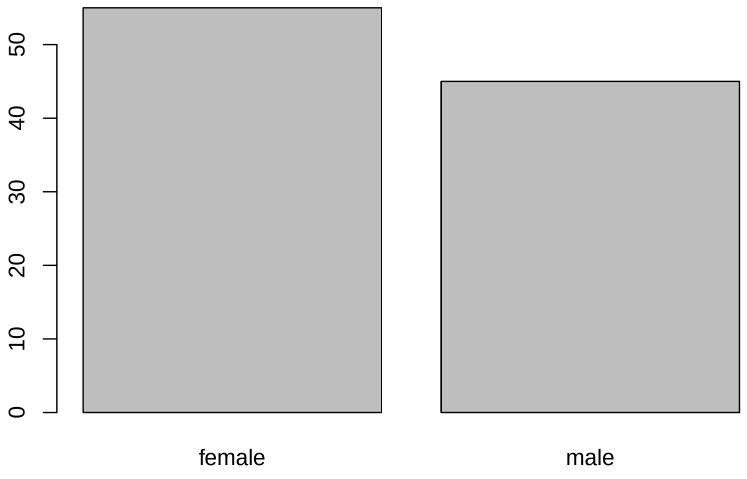
4.1.2 Barplots

Barplots can be generated in several ways:

- By passing a factor to `plot()` - it will generate a barplot of level frequencies. This is a shorthand for `barplot(table(some_factor))`.
- By using `barplot()`. The advantage of this is that accepts some graphical parameters that are not relevant and accepted by `plot()`, such as `beside =`, `height =`, `width =` and others (type `?barplot` to see all).

Here is an example:

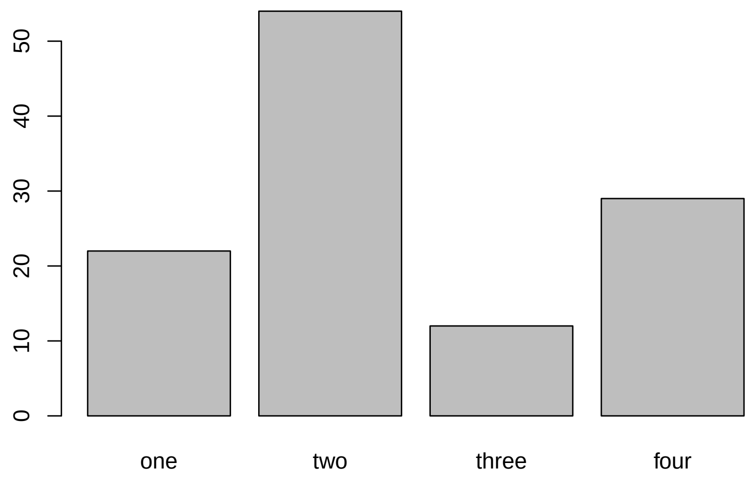
```r
persons <- as.factor(sample(c("male", "female"), size = 100, replace = T))
plot(persons)
```



`barplot()` with a vector

The function `barplot()` can be called with a vector specifying the bar heights (frequencies), or a `table` object.

```
frequencies <- c(22, 54, 12, 29)
barplot(frequencies, names = c("one", "two", "three", "four"))
```

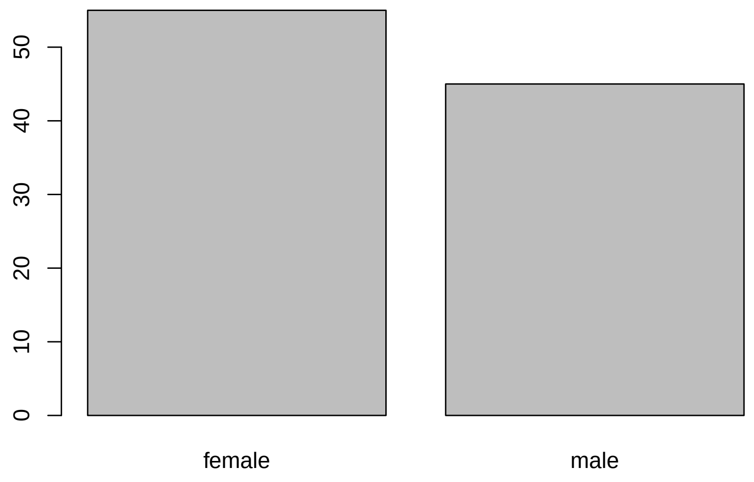


With a table object:

```
table(persons)
```

```
## persons
## female   male
##     55     45
```

```
barplot(table(persons))
```



`barplot()` with a 2D table object

Suppose you have this data:

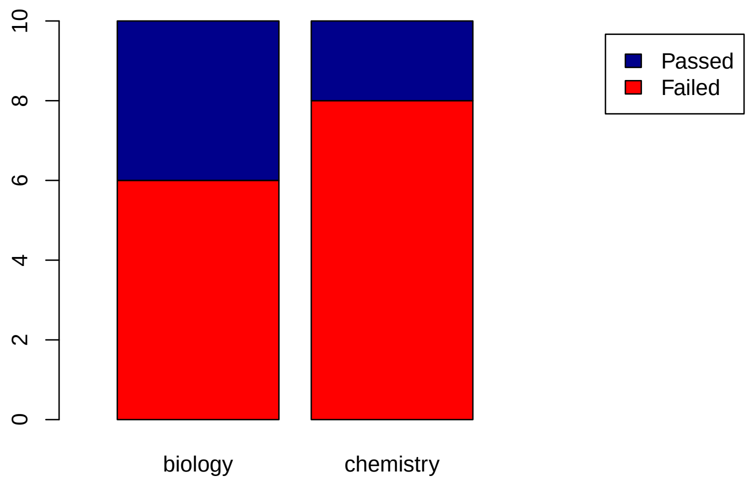
```r
set.seed(1234)
course <- rep(c("biology", "chemistry"), each = 10)
passed <- sample(c("Passed", "Failed"), size = 20, replace = T)
tbl <- table(passed, course) # the order matters!
tbl
```

```
##         course
## passed   biology chemistry
##   Failed       6         8
##   Passed       4         2
```

The `set.seed(1234)` makes the *sampling* reproducible, although that sounds really unlogical. Discussing ***pseudorandom*** sampling is not within the scope of this course however.

You can create a ***stacked bar chart*** like this.

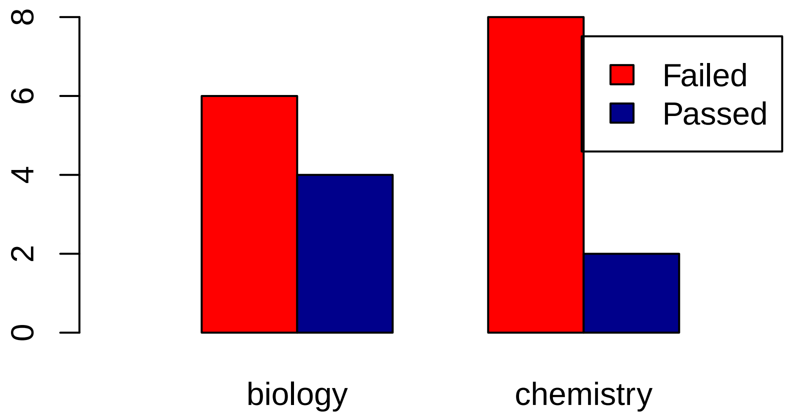
```r
barplot(tbl,
        col = c("red", "darkblue"),
        xlim = c(0, ncol(tbl) + 2),
        legend = rownames(tbl))
```



The `xlim =` setting is a trick to get the legend beside the plot.

Using the `beside = TRUE` argument, you get the bars ***side by side***:

```r
barplot(tbl,
        col=c("red", "darkblue"),
        beside = TRUE,
        xlim=c(0, ncol(tbl)*2 + 3),
        legend = rownames(tbl))
```

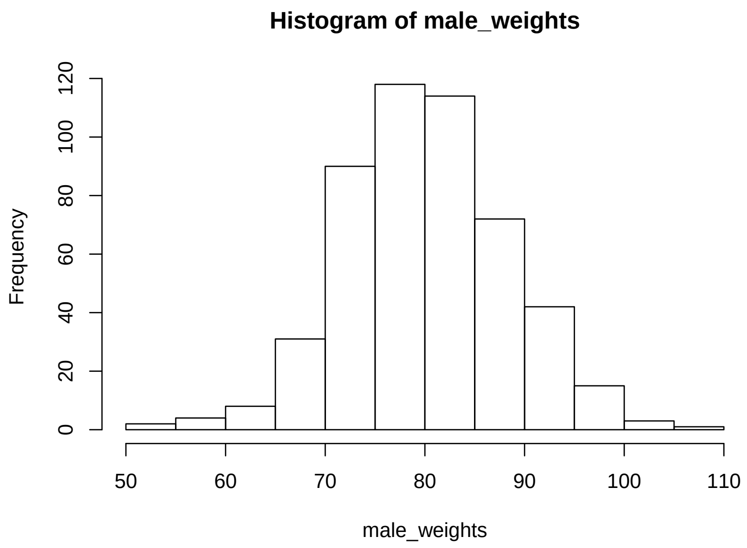


Later, we'll see another data structure to feed to barplot: the matrix.

4.1.3 Histograms

Histograms help you visualise the distribution of your data.

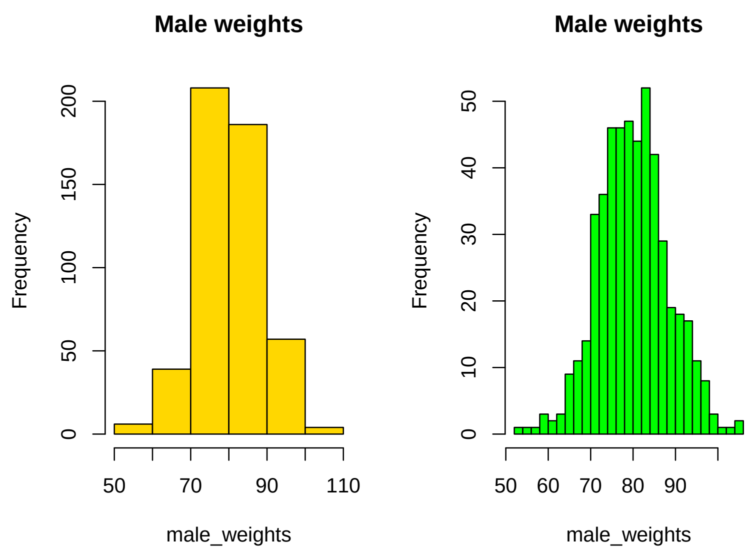
```
male_weights <- c(rnorm(500, 80, 8)) ## create 500 random numbers around 80
hist(male_weights)
```



Using the `breaks` argument, you can adjust the bin width. Always explore this option when creating histograms!

```
par(mfrow = c(1, 2)) # make 2 plots to sit side by side
hist(male_weights, breaks = 5, col = "gold", main = "Male weights")
```

```r
hist(male_weights, breaks = 25, col = "green", main = "Male weights")
```

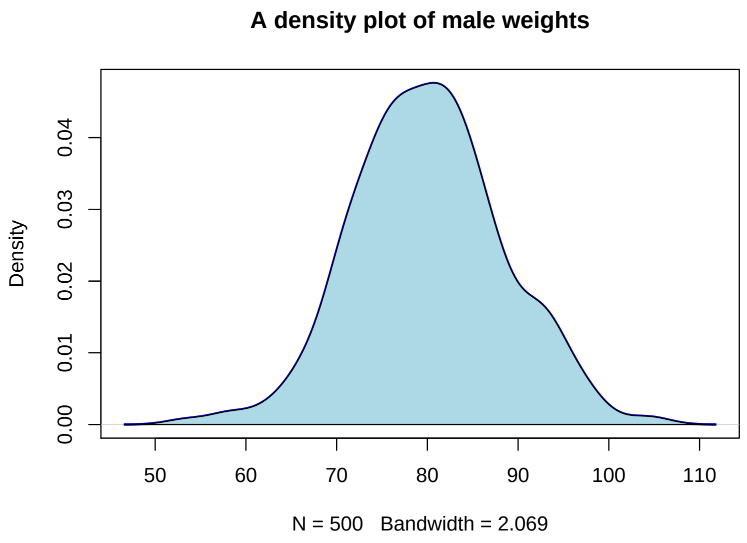


If you want a more detailed

4.1.4 Density plot as alternative to `hist()`

When you want a bit more fine-grained view of the distribution you can use a plot of a density function; by adding a `polygon()` you can even have some nice shading under the line:

```r
plot(density(male_weights),
     main = "A density plot of male weights",
     col = "blue", lwd = 2)
polygon(density(male_weights), col="lightblue")
```



4.1.5 Boxplots

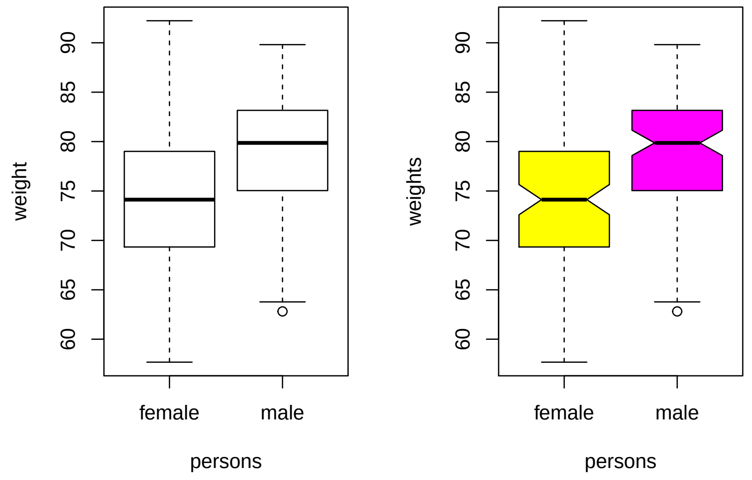
This is the last of the basic plot types. A boxplot is a visual representation of the *5-number summary* of a numeric variable: minimum, maximum, median, first and third quartile.

```
persons <- rep(c("male", "female"), each = 100)
weights <- c(rnorm(100, 80, 6), rnorm(100, 75, 8))
#print 6-number summary (5-number + mean)
summary(weights[persons == "female"])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   57.68   69.36   74.12   73.99   78.97   92.23
```

Boxplots tell the same story as histograms, but are less precise. however, they are excellent when you want to show a series of subsets split over some variable.

```
par(mfrow = c(1, 2)) # make 2 plots to sit side by side
# create boxplots of weights depending on sex
boxplot(weights ~ persons, ylab = "weight")
boxplot(weights ~ persons, notch = TRUE, col = c("yellow", "magenta"))
```

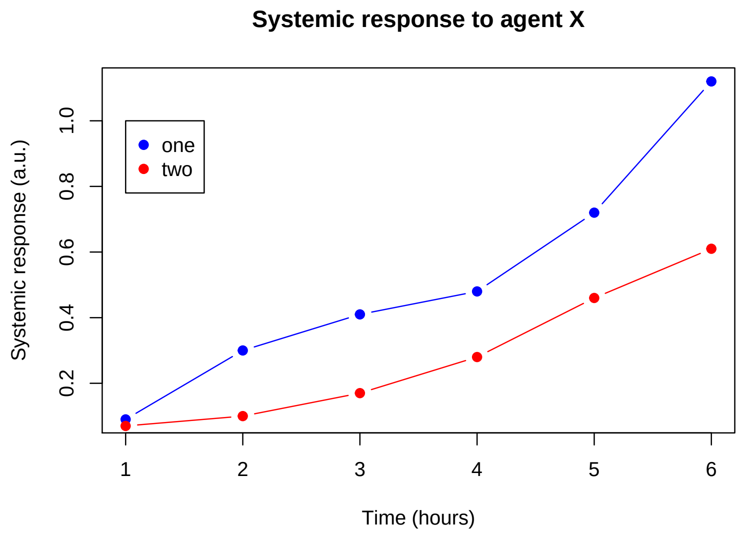


Use `varwidth = TRUE` when you want to visualize the difference in group sizes.

4.1.6 Adding more data and a legend

When you have more than one data series to plot, add them using the function `points()`. You call this function *after* you created the primary plot. Since there are multiple lines you will also need a legend.

```
response2 <- c(0.07, 0.10, 0.17, 0.28, 0.46, 0.61)
plot(x = time, y = response, pch = 19, type = "b",
     xlab = "Time (hours)", ylab = "Systemic response (a.u.)",
     main = "Systemic response to agent X", col = "blue")
points(x = time, y = response2, col = "red", pch = 19, type = "b")
legend(x = 1, y = 1.0, legend = c("one", "two"), col = c("blue", "red"), pch = 19)
```



The `legend()` function is *very* versatile. Have a look at the docs! In its most basic form you pass it a position (x and y), series names, colors and plot character.

4.1.7 Helper lines and `lm()`

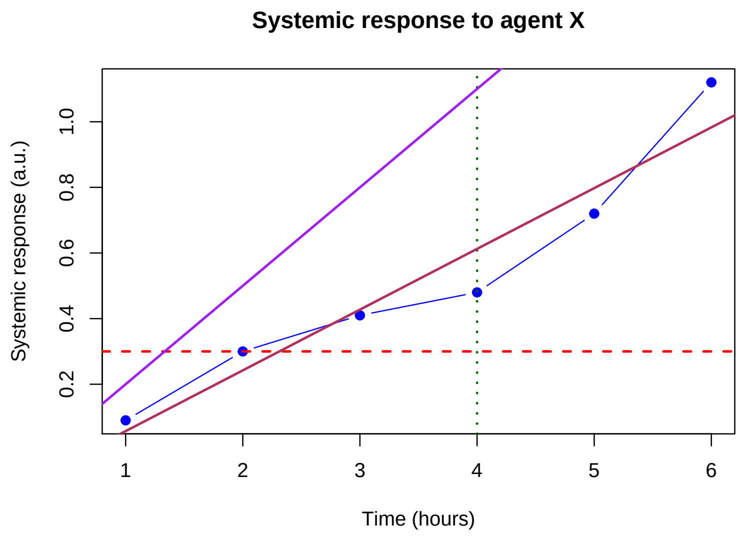
Adding helper lines can be used to aid your reader in grasping and interpreting your data story. Use the function `abline()` for this.

There are four types of helper lines you might want to add to a figure:

- A horizontal line with `h =`: indicate some y-threshold
- A vertical line with `v =`: indicate x-threshold or mean or some other statistic
- A line with an intercept (`a =`) and a slope (`b =`): often used to indicate some expected response, or diagonal x = y
- A linear model, determined with the `lm()` function. The linear model object actually contains an intercept and a slope value which is taken by `abline()`.

In the following plot, these four basic helper lines are demonstrated:

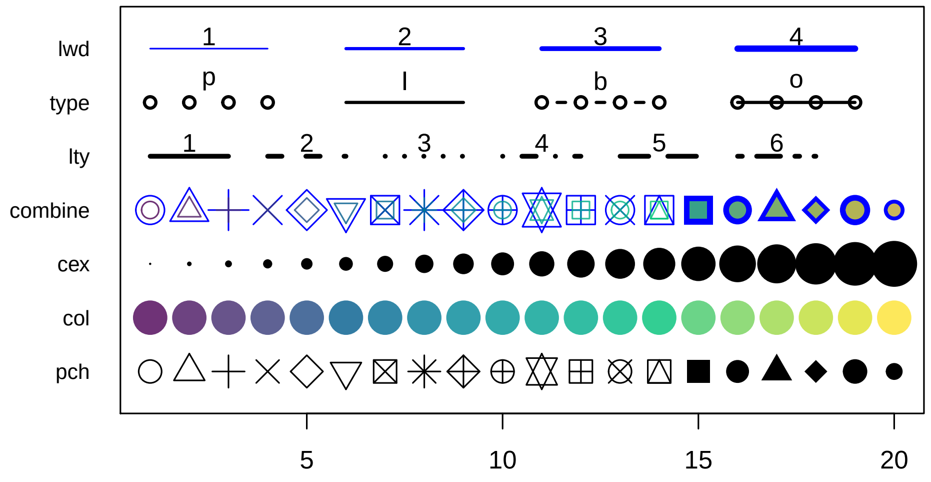
```r
plot(x = time, y = response, pch = 19, type = "b",
     xlab = "Time (hours)", ylab = "Systemic response (a.u.)",
     main = "Systemic response to agent X", col = "blue")
#horizontal line
abline(h = 0.3, lty = 2, lwd = 2, col = "red")
#vertical line
abline(v = 4, lty = 3, lwd = 2, col = "darkgreen")
#line with slope
abline(a = -0.1, b = 0.3, lwd = 2, col = "purple")
#linear model
abline(lm(response ~ time),  lwd = 2, col = "maroon")
```



4.2 Graphical parameters to `plot()`

There are *many* parameters that can be passed to the plotting functions. Here is a small sample and their possible values.

```
series <- 1:20
plot(0, 0, xlim=c(1,20) , ylim=c(0.5, 7.5), col="white" , yaxt="n" , ylab="" , xlab=""

# the rainbow() function gives a nice palette across all colors
# or use hcl.colors() to specify another palette
# use  hcl.pals() to get an overview of available pallettes
colors = hcl.colors(20, alpha = 0.8, palette = 'viridis')

#pch
points(series, rep(1, 20), pch = 1:20, cex = 2)
#col
points(series, rep(2, 20), col = colors, pch = 16, cex = 3)
#cex
points(series, rep(3, 20), col = "black" , pch = 16, cex = series * 0.2)

#overlay to create new symbol
points(series, rep(4, 20), pch = series, cex = 2.5, col = "blue")
points(series, rep(4, 20), pch = series, cex = 1.5, col = colors)

#lty
for (i in 1:6) {
    points(c(-2, 0) + (i * 3), c(5, 5), col = "black", lty = i, type = "l", lwd = 3)
    text((i * 3) - 1, 5.25 , i)
}
#type and lwd
for (i in 1:4) {
    #type
    points(c(-4, -3, -2, -1) + (i * 5), rep(6, 4),
           col = "black", type = c("p","l","b","o")[i], lwd=2)
    text((i * 5) - 2.5, 6.4 , c("p","l","b","o")[i] )
    #lwd
    points(c(-4, -3, -2, -1) + (i * 5), rep(7, 4), col = "blue", type = "l", lwd = i)
    text((i * 5) - 2.5, 7.23, i)
}
#add axis
axis(side = 2, at = c(1, 2, 3, 4, 5, 6, 7),
    labels = c("pch" , "col" , "cex" , "combine", "lty", "type" , "lwd" ),
    tick = FALSE, col = "black", las = 1, cex.axis = 0.8)
```



Chapter 5

Complex Datatypes

TO BE PORTED FROM PRESENTATION

Chapter 6

Functions

TO BE PORTED FROM PRESENTATION

Chapter 7

Scripting

TO BE PORTED FROM PRESENTATION

Chapter 8

Dataframe manipulations

TO BE PORTED FROM PRESENTATION

Chapter 9

Exercises

This chapter only contains exercises. The solutions are in the next chapter which has a numbering parallel to this one.

9.1 Basic R

9.1.1 Math in the console

In the console, calculate the following:

$31 + 11$

$66 - 24$

$\frac{126}{3}$

12^2

$\sqrt{256}$

$\frac{3*(4+\sqrt{8})}{5^3}$

9.1.2 First look at functions

1. View the help page for `paste()`. There are two variants of this function.
    - Which? And what is the difference between them?
    - Use both variants to generate exactly this message `"welcome to R"` from these arguments: `"welcome ", "to ", "R"`
2. What does the `abs` function do?
    - What is returned by `abs(-20)` and what is `abs(20)`?

3. What does the `c` function do?

- What is the difference in returned value of `c()` when you combine either `1`, `3` and `"a"` as arguments , or `1`, `2` and `3`?

9.1.3 Variables

Create three variables with the given values - x=20, y=10 and z=3. Next, calculate the following with these variables:

1. $x + y$
2. x^z
3. $q = x \times y \times z$
4. $\sqrt{q}$
5. $\frac{q}{\pi}$ (pi is simply pi in R)
6. $\log_{10}(x \times y)$

Plotting rules

With all plots, take care to adhere to the rules regarding titles and other decorations. Tip: the site Quick-R has nice detailed information with examples on the different plot types and their configuration. Especially the section on plotting is helpful for these assignments.

9.1.4 Stair walking and heart rate

The vectors below hold data for a staircase walking experiment. A subject of normal weight and height was asked to ascend a (long) stairs wearing a heart-rate monitor. The subjects' heart was registered for different step heights. Create a **line plot** showing the dependence of heart rate (y axis) on stair height (x axis).

```r
#number of steps on the stairs
stair_height <- c(0, 5, 10, 15, 20, 25, 30, 35)
#heart rate after ascending the stairs
heart_rate <- c(66, 65, 67, 69, 73, 79, 86, 97)
```

9.1.5 More subjects

The experiment from the previous question was extended with three more subjects. One of these subjects was like the first of normal weight, whereas the two others were obese. The data are given below. Create a single **scatter plot** with connector lines between the points showing the data for all four subjects. Give the normal-weighted subjects a green line and symbol and the obese subjects a red line and symbol.
You can add new data series to a plot by using the `points(x, y)` function. Use the `ylim()` function to adjust the Y-axis range.

```r
#number of steps on the stairs
stair_height <- c(0, 5, 10, 15, 20, 25, 30, 35)
#heart rates for subjects with normal weight
heart_rate_1 <- c(66, 65, 67, 69, 73, 79, 86, 97)
heart_rate_2 <- c(61, 61, 63, 68, 74, 81, 89, 104)
#heart rates for obese subjects
heart_rate_3 <- c(58, 60, 67, 71, 78, 89, 104, 121)
heart_rate_4 <- c(69, 73, 77, 83, 88, 96, 102, 127)
```

9.1.6 Chickens on a diet

The body weights of chicks were measured at birth and every second day thereafter until day 20. They were also measured on day 21. In the experiment there were four groups of chicks on different protein diets. Here are the data for the first four chicks. Chick one and two were on diet 1 and chick three and four were on diet 2. Create a single line plot showing the data for all four chicks. Give each chick its own color.

```r
# chick weight data
time <- c(0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 21)
chick_1 <- c(42, 51, 59, 64, 76, 93, 106, 125, 149, 171, 199, 205)
chick_2 <- c(40, 49, 58, 72, 84, 103, 122, 138, 162, 187, 209, 215)
chick_3 <- c(42, 53, 62, 73, 85, 102, 123, 138, 170, 204, 235, 256)
chick_4 <- c(41, 49, 61, 74, 98, 109, 128, 154, 192, 232, 280, 290)
```

9.1.7 Chicken bar plot

With the data from the previous question, create a bar plot of the maximum weights of the chicks.

9.1.8 Discoveries

The R language comes with a wealth of datasets for you to use as practice materials. We will see several of these. One of these datasets is The Time-Series dataset called `discoveries` holding the numbers of "great" inventions and scientific discoveries in each year from 1860 to 1959. Type its name in the console to see it. Create plot(s) answering these questions:

A

What is the number of discoveries per year? Use the `barplot()` and `table()` functions for this.

B

What is the 5-number summary of discoveries per year?

C

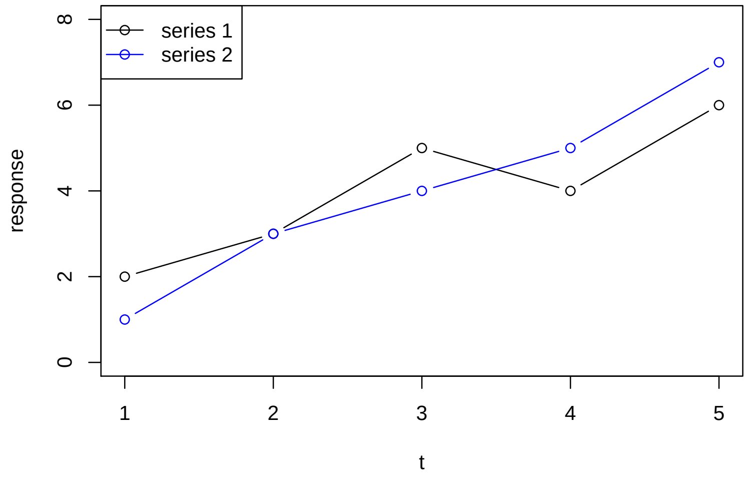
What is the trend over time for the numbers of discoveries per year?

PS: This is actually not a simple vector but a vector with some time-related attributes. It is called a Time-Series (a `ts` class), but this does not really matter for this assignment.

9.1.9 Lung cancer

The R datasets package has three related timeseries datasets relating to lung cancer deaths. These are `ldeaths`, `mdeaths` and `fdeaths` for total, male and female deaths respectively. Create a line plot showing the monthly mortality holding all three of these datasets. Use the `legend()` function to add a legend to the plot, as demonstrated in this example:

```r
t <- 1:5
y1 <- c(2, 3, 5, 4, 6)
y2 <- c(1, 3, 4, 5, 7)
plot(t, y1, type = "b", ylab = "response", ylim = c(0, 8))
points(t, y2, col = "blue", type = "b")
legend("topleft", legend = c("series 1", "series 2"), col = c("black", "blue"), pch = 
```



A

Create the mentioned line plot. Do you see trends and/or patterns and if so, can you explain these?

B

Create a combined boxplot of the three time-series. Are there outliers? If so, can you figure out when this occurred?

9.2 Complex datatypes

This section serves you some datatype challenges.

9.2.1 Creating factors

A

Given this vector:

```
animal_risk <- c(2, 4, 1, 1, 2, 4, 1, 4, 1, 1, 2, 1)
```

and these possible levels: 1: harmless 2: risky 3: dangerous 4: deadly

Create a factor from this data and then barplot the result.

B

Given this data, a simulation of wealth distribution of "poor", "middle class", "wealthy" "rich:

```
set.seed(1234)
wealth_male <- sample(x = letters[1:4],
                      size = 1000,
                      replace= TRUE,
                      prob = c(0.7, 0.17, 0.12, 0.01))
wealth_female <- sample(x = letters[1:4],
                      size = 1000,
                      replace= TRUE,
                      prob = c(0.8, 0.15, 0.497, 0.003))
```

Create a factor from these two and report the cumulative percentage of its individual levels starting at the most abundant level, combined for male and female. Hint: use `table()` and `prop.table()`.

Next, create a side-by-side barplot of this data. Don't forget the legend!

9.2.2 A dictionary with a named vector

Almost all programming languages know the (hash)map / dictionary data structure storing so-called "key-and-value" pairs. They make it possible to "look up" the value belonging to a "key". That is where the term dictionary comes from. A dictionary holds keys (the words) and their meaning (values). R does not have a dictionary type but you could make a dict-like structure using a ***vector with named elements***. Here follows an example.

If I wanted to create and use a DNA codon translation table, and use it to translate a piece of DNA, I could do something like what is shown below (there are only 4 of the 64 codons included). See if you can figure out what is going on there

```r
## define codon table as named vector
codons <- c("Gly", "Pro", "Lys", "Ser")
names(codons) <- c("GGA", "CCU", "AAA", "AGU")

## the DNA to translate
my_DNA <- "GGACCUAAAAGU"
my_prot <- ""
## iterate the DNA and take only every position
for (i in seq(1, nchar(my_DNA), by=3)) {
    codon <- substr(my_DNA, i, i+2);
    my_prot <- paste(my_prot, codons[codon])
}
print(my_prot)
```

```
## [1] " Gly Pro Lys Ser"
```

A

Make a modified copy of this code chunk in such a way that no spaces are present between the amino acid residues (use help on `paste()` to figure this out) and that single-letter codes of amino acids are used instead of three-letter codes.

B

[**Challenge**] Here is a vector called `nuc_weights`. It holds the weights for the nucleotides A, C, G and U respectively. Make it a named vector, iterate `my_DNA` from the above code chunk and calculate its molecular weight.

```r
nuc_weights <- c(491.2, 467.2, 507.2, 482.2)
```

9.2.3 airquality

The `airquality` dataset is also one of the datasets included in the `datasets` package. We'll explore this for a few questions.

A

Create a scatterplot of Temperature as a function of Solar radiation. Is there, as you might naively expect, a strong correlation? You could use `cor.test()` to find out. Add a linear model using `lm()` to extend your plot.

B

Create a boxplot-series of `Temp` as a function of `Month` (use `?boxplot` to find out how this works). What appears to be the warmest month?

C

What date (day/month) has the lowest recorded temperature? Which the highest? Please give temperature values in Celsius, not Fahrenheit! (Yes, this is an extra challenge!)

D

Create a histogram of the wind speeds, and add a thick blue vertical line for the value of the mean and a fat red line for the median (use `abline()` for this).

E

Use the `pairs()` function with argument `panel = panel.smooth` to plot all pairwise correlations between Ozone, Solar radiation, Wind and Temperature. Which pair shows the strongest correlation in your opinion? Verify this using the `cor()` function after removing incomplete cases. Create a separate well annotated scatterplot of this pair.

9.2.4 Bird observations

You will explore a bird observation dataset, downloaded from GOLDEN GATE AUDUBON SOCIETY. This file lists bird observations collected by this bird monitoring group in the San Francisco Bay Area. I already cleaned it a bit and placed it here: data/Observations-Data-2014.csv.

You can download it as follows:

```
file_name <- "Observations-Data-2014.csv"
remote_url <- paste0("https://raw.githubusercontent.com/MichielNoback/davur1_gitbook/master/data/

download.file(url = remote_url, destfile = file_name)
```

Load the observation data into R and assign it to a variable called `bird_obs`.

From here on, it is assumed that you have the dataframe `bird_obs` loaded. This series of exercises deals with cleaning and transforming data, and exploring a cleaned dataset using basic plotting techniques and descriptive statistics.

A

First, explore the raw data as they are.

- What data on bird observations were recorded (i.e. what kind of variables do you have)?
- What did R do to the original column names?
- Are all column names clear to you?

B

How many bird observations were recorded?

C

The column holding observation "Number" is actually not a number. Into what type has R converted it?

D

Convert the “Number” column into an integer column using `as.integer()`, but assign it to a new column called “Count” (i.e. do not overwrite the original values). Compare the first 50 values or so of these two columns. What happened to the data? Is this OK?

E

The previous question has shown that converting factors to numbers is a bit dangerous. It is often easiest to convert characters to numbers. The best way to do this is by using the `as.is = c(<column indices>)` argument for the `read.table()` function.

So, which columns should be loaded as real factor data and which as plain character data? Use `read.table()` and the `as.is` argument to reload the data, and then transform the `Number` column to integer again as `Count`.

F

Compare the first 50 values of the Number and Count columns again. Has the conversion succeeded? How many `Number` values could not be transformed into an integer value? Hint: use `is.na()`

G

Explore the sighting counts:

- What is the maximum number of birds in a single sighting? (Use max() and which() or is.na() to solve this)
- What is the mean sighting count
- What is the median of the sighting count

H

Is the `Count` variable a normal distributed value? You can use `hist(...)`, `table()` or `plot(density(...))` to explore this further.

I

Explore the species constitution:

- How many different species were recorded?
- How many genera do they constitute?
- What species from the genus “Puffinus” have been observed?

Hint: use the function `unique()` here.

J [Challenge]
This is a challenge exercise for those who like to grind their brains! Think of a strategy to “rescue” the NAs that appear after transforming “Number” to “Count”. Hint: use `gsub()` or`grep()`

9.3 Regular Expressions

9.3.1 Restriction enzymes

A

The restriction enzyme PacI has the recognition sequence “TTAATTAA”. Define (at least) three alternative regex patterns that will catch these sites.

B

The restriction enzyme SfiI has the recognition sequence “GGCCNNNNNGGCC”. Define (at least) three alternative regex patterns that will catch these sites.

9.3.2 Prosite Patterns

A

The Prosite pattern PS00211 (ABC-transporter-1; https://prosite.expasy.org/PS00211) has the pattern:
“[LIVMFYC]-[SA]-[SAPGLVFYKQH]-G-[DENQMW]-[KRQASPCLIMFW]-[KRNQSTAVM]-[KRACLVM]-[LIVMFYPAN]-{PHY}-[LIVMFW]-[SAGCLIVP]-{FYWHP}-{KRHP}-[LIVMFYWSTA].” Translate it into a regex pattern. Info on the syntax is here: https://prosite.expasy.org/prosuser.html#conv_pa

B

The Prosite pattern PS00018 (EF-hand calcium-binding domain; https://prosite.expasy.org/PS00018) has the pattern: “D-{W}-[DNS]-{ILVFYW}-[DENSTG]-[DNQGHRK]-{GP}-[LIVMC]-[DENQSTAGC]-x(2)- [DE]-[LIVMFYW].” Translate it into a regex pattern.

You could exercise more by simply browsing Prosite. Test your pattern by fetching the proteins referred to within the Prosite pattern details page.

9.3.3 Fasta Headers

The fasta sequence format is a very common sequence file format used in molecular biology. It looks like this (I omitted most of the actual protein sequences for better representation):

```
>gi|21595364|gb|AAH32336.1| FHIT protein [Homo sapiens]
MSFRFGQHLIK...ALRVYFQ
>gi|15215093|gb|AAH12662.1| Fhit protein [Mus musculus]
MSFRFGQHLIK...RVYFQA
>gi|151554847|gb|AAI47994.1| FHIT protein [Bos taurus]
MSFRFGQHLIK...LRVYFQ
```

As you can see there are several distinct elements within the Fasta ***header*** which is the description line above the actual sequence: one or more database identification strings, a protein description or name and an organism name. Study the format - we are going to extract some elements from these fasta headers using the `stringr` package. Install it if you don't have it yet.

Here is a small example:

```r
library(stringr)
hinfII_re <- "GA[GATC]TC"
sequences <- c("GGGAATCC", "TCGATTCGC", "ACGAGTCTA")
str_extract(string = sequences,
            pattern = hinfII_re)
```

```
## [1] "GAATC" "GATTC" "GAGTC"
```

Function `str_extract()` simply extracts the exact match of your regex (shown above). On the other hand, function `str_match()` supports ***grouping capture*** through bounding parentheses:

```r
phones <- c("+31-6-23415239", "+49-51-55523146", "+31-50-5956566")
phones_re <- "\\+(\\d{2})-(\\d{1,2})" #matching country codes and area codes
matches <- str_match(phones, phones_re)
matches
```

```
##      [,1]     [,2] [,3]
## [1,] "+31-6"  "31" "6" 
## [2,] "+49-51" "49" "51"
## [3,] "+31-50" "31" "50"
```

Thus, each set of parentheses will yield a column in the returned matrix. Simply use its column index to get that result set:

```r
matches[, 2] ##the country codes
```

```
## [1] "31" "49" "31"
```

Now, given the fasta headers in ./data/fasta_headers.txt which you can simply load into a character vector using `readLines()`, extract the following.

A

- Extract all complete organism names.
- Extract all species-level organism names (omitting subspecies and strains etc).

B

Extract all ***first*** database identifiers. So in this header element `>gi|224017144|gb|EEF75156.1|` you should extract only `gi|224017144`

C

Extract all protein names/descriptions.

9.4 Scripting

This section serves you some exercises that will help you improve your function-writing skills.

9.4.1 Illegal reproductions

As an exercise, you will re-invent the wheel here for some statistical functions.

The mean

Create a function, `my_mean()`, that duplicates the R function `mean()`, i.e. calculates and returns the mean of a vector of numbers, without actually using `mean()`.

Standard deviation

Create a function, `my_sd()`, that duplicates the R function `sd()`, i.e. calculates and returns the standard deviation of a vector of numbers, without actually using `sd()`.

Median

[**Challenge**] Create a function, `my_median()`, that duplicates the R function `median()`, i.e. calculates and returns the median of a vector of numbers. This is actually a bit harder than you might expect. Hint: use the `sort()` function.

9.4.2 Interquantile ranges

Create a function that will calculate a custom "interquantile range". The function should accept three arguments: a numeric vector, a lower quantile and an upper quantile. It should return the difference (range) between these two quantile values. The lower quantile should default to 0 and the higher to 1, thus returning `max(x)` minus `min(x)`. The function therefore has this "signature":

```
interquantile_range <- function(x, lower = 0, higher = 100) {}
```

Perform some tests on the arguments to make a robust method: are all arguments numeric?

To test you method, you can compare `interquantile_range(some_vector, 0.25, 0.75)` with `IQR(some_vector)` - they should be the same.

9.4.3 Vector distance

Create a function, `distance(p, q)`, that will calculate and return the Euclidean distance between two vectors of equal length. A numeric vector can be seen as a point in multidimensional space. Euclidean distance is defined as

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

Where p and q are the two vectors and n the length of the two vectors.
You should first perform a check whether the two vectors are of equal length and both of type `numeric` or `integer`. If not, the function should abort with an appropriate error message.

Other distance measures

Extend the function of the previous assignment in such a way that a third argument is accepted, `method =`, which defaults to "euclidean". Other possible distance measures are "Manhattan" (same as "city block" and "taxicab") and Pearson correlation. Look the equations for these up in Wikipedia or some other place.

9.4.4 G/C percentage of DNA

[**Challenge XL**] Create a function, `GC_perc()`, that calculates and returns the GC percentage of a DNA or RNA sequence. Accept as input a sequence and a flag `-strict-` indicating whether other characters are accepted than core DNA/RNA (GATUC). If `strict = FALSE`, the percentage of other characters should be reported using a `warning()` call. If `strict = TRUE`, the function should terminate with an error message. Use `stop()` for this. `strict` should default to `TRUE`. NOTE, usage of `strict` can complicate things, so start with the core functionality! You can use `strsplit()` or `substr()` to get hold of individual sequence characters.

9.5 Function apply and its relatives

In this section you will encounter some exercises revolving around the different flavors of apply.

9.5.1 Whale selenium

On the course website under Resources you will find a link to file `whale_selenium.txt`. You could download it into your working directory manually or use `download.file()` to obtain it. However, there is a third way to get its contents without actually downloading it as a local copy. You can read it directly using `read.table()` as shown here.

```
whale_sel_url <- "https://raw.githubusercontent.com/MichielNoback/davur1/gh-pages/exer
whale_selenium <- read.table(whale_sel_url,
    header = T,
    row.names = 1)
```

Note: when you are going to load a file many times it is probably better to store a local copy.

A

Report the means of both columns using `apply()`.

B

Report the standard deviation of both columns, using `apply()`

C

Report the standard error of the mean of both columns, using `apply()` The SEM is calculated as

$$\frac{sd}{\sqrt{n}}$$

where sd is the sample standard deviation and n the number of measurements. You should create the function calculating this statistic yourself.

D

Using `apply()`, calculate the ratio of Se_{tooth}/Se_{liver} and attach it to the `whale_selenium` dataframe as column `ratio`. Create a histogram of this ratio.

E

Using `print()` and `paste()`, report the mean and the standard deviation of the ratio column, but do this with an inline expression, e.g. an expression embedded in the R markdown paragraph text.

9.5.2 ChickWeight

This exercise revolves around the `ChickWeight` dataset of the built-in `datasets` package.

A

Report the number of chickens used in the experiment.

B

Use `aggregate()` to get the mean weight of the chickens for the different Diets.

C

Use `coplot()` to plot a panel with weight as function of Time, split over Diet.

D

Add a column called `weight_gain` to the dataframe holding values for the weight gain since the last measurement. Take special care with rows marking the boundaries between individual chickens! You could consider using a traditional for loop here. In the next course, we'll see a more efficient way of doing this.

E

Split the `weight_gain` column on Diet and report the mean, median and standard deviation for each diet. If you were not successful in the previous question, load and attach the data from file `ChickWeight_weight_gain.Rdata` downloadable from https://github.com/MichielNoback/davur1_gitbook/raw/master/data/ChickWeight_weight_gain.Rdata. You can use this code chunk for downloading and loading the data into variable `stored_weight_gain`. Don't forget to attach the column to the data frame!

```
local_file <- "ChickWeight_weight_gain.Rdata"
download.file(paste0("https://github.com/MichielNoback/davur1_gitbook/raw/master/data/
load(local_file)
```

F

Create a (single-panel) boxplot for weight gain, split over Diet. Hint: read the `boxplot()` help page!

9.5.3 Food constituents

The food constituents dataset holds information on ingredients for different foods. Individual foods are simply marked with an id.

A

Load the data and report the different food categories (`Type`). Also report the numbers of entries for each Type.

B

What is the mean energy content of chocolate foods?

C

What is the food category with the highest mean fat content?

D

What food category has the highest mean energy content, and which has the lowest?

E

[**Challenge**] Create a boxplot showing the difference in sugar content between drink and solid food.

F

Assuming both unsaturated fats and sugar are bad for you, what food category do you consider the worst? Think of a means to answer this, explain it and carry it out.

9.5.4 Bird observations revisited

This exercise revisits the bird observations dataset. You can download it here. (Re)load the dataset.

A

Report the number of observations per `County`. Use both a textual as a barplot representation. With the barplot, you should order the bars according to observation numbers.

B

Report the number of observations per `Observer.1` but only for observers with more than 10 observations, ordered from high to low observation count. Use `order()` to achieve this.

C

Which observer has the highest number of observations listed (and how many is that)?

D

Report the different observed species (using `Common.name`) for each genus. [**Challenge**] Report only the 5 Genera with the highest number of observed species.

E

[**Challenge**] Create a Dataframe holding the number of birds per day (use Date.start) and plot it with date on the x-axis and number of birds on the y-axis. Hint: use `as.Date()` to convert the character date to a real date field. See this page how you can do that Date Values.

Chapter 10

Exercise solutions

10.1 Basic R

10.1.1 Math in the console

$31 + 11$

$66 - 24$

$\frac{126}{3}$

12^2

$\sqrt{256}$

$\frac{3*(4+\sqrt{8})}{5^3}$

```r
31 + 11
66 - 24
126 / 3
12^2
256**0.5
(3 * (4 + 8^0.5))/(5^3)
```

```
## [1] 42
## [1] 42
## [1] 42
## [1] 144
## [1] 16
## [1] 0.1638823
```

10.1.2 First look at functions

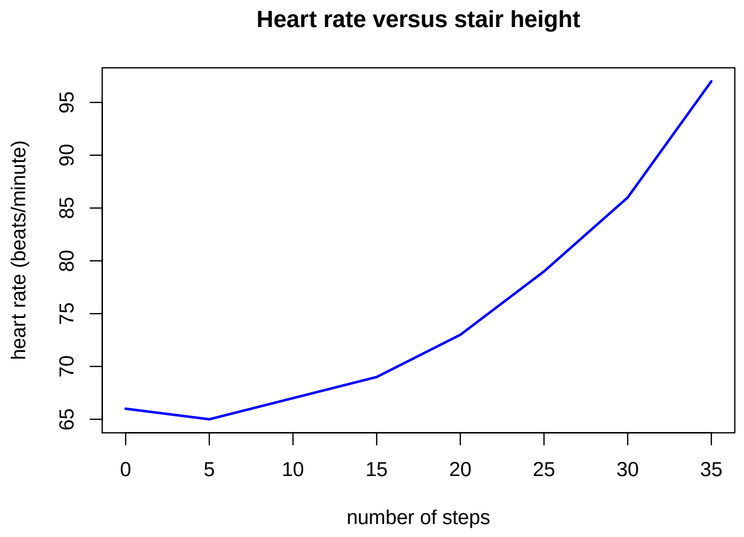
`[NO SOLUTION YET]`

Plotting rules

Since everything needs to be done in a (corona virus induced) rush, plots may be (far) from perfect. Sorry about that.

10.1.3 Stair walking and heart rate

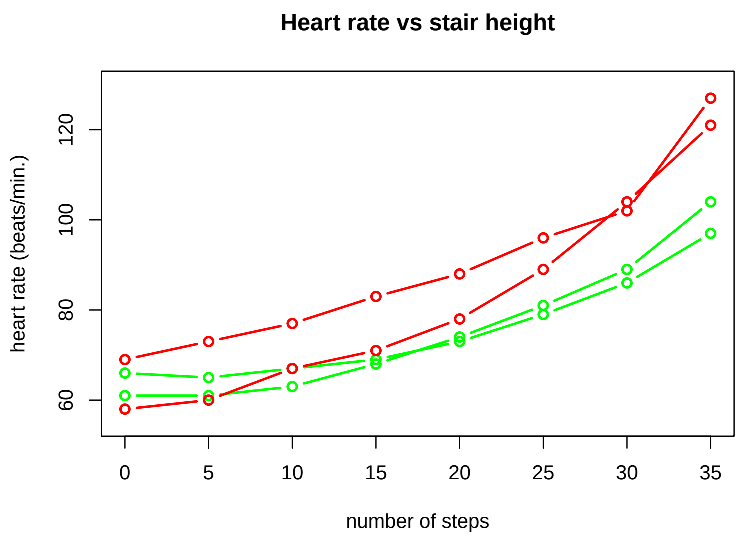
```
#number of steps on the stairs
stair_height <- c(0, 5, 10, 15, 20, 25, 30, 35)
#heart rate after ascending the stairs
heart_rate <- c(66, 65, 67, 69, 73, 79, 86, 97)
plot(heart_rate ~ stair_height,
      main = "Heart rate versus stair height",
      xlab = "number of steps",
      ylab = "heart rate (beats/minute)",
      type = "l",
      lwd = 2,
      col = "blue")
```



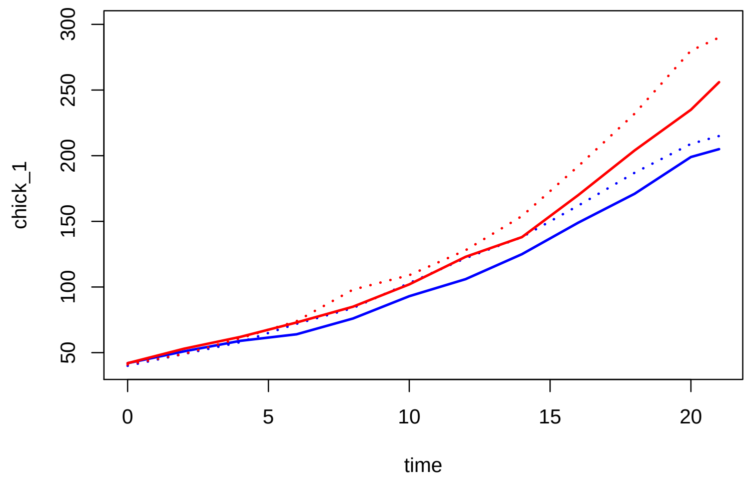
10.1.4 More subjects

```
#number of steps on the stairs
stair_height <- c(0, 5, 10, 15, 20, 25, 30, 35)
#heart rates for subjects with normal weight
heart_rate_1 <- c(66, 65, 67, 69, 73, 79, 86, 97)
heart_rate_2 <- c(61, 61, 63, 68, 74, 81, 89, 104)
#heart rates for obese subjects
heart_rate_3 <- c(58, 60, 67, 71, 78, 89, 104, 121)
```

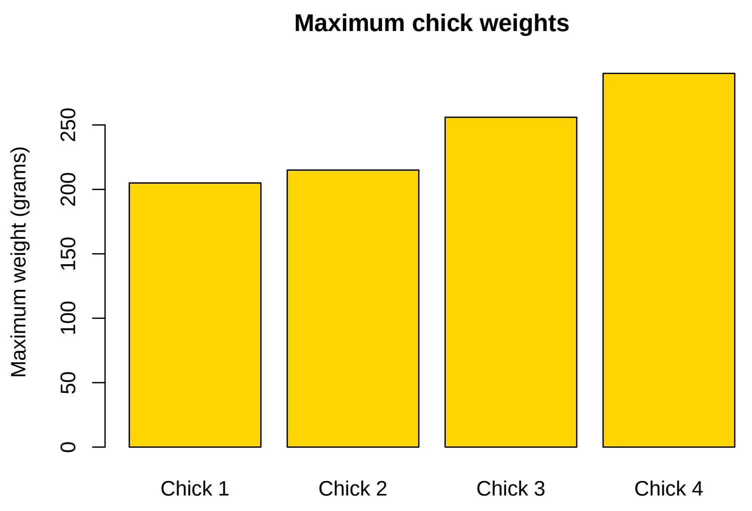
```
heart_rate_4 <- c(69, 73, 77, 83, 88, 96, 102, 127)
plot(x = stair_height,
    y = heart_rate_1,
    main = "Heart rate vs stair height",
    xlab = "number of steps",
    ylab = "heart rate (beats/min.)",
    type = "b",
    lwd = 2,
    col = "green",
    ylim = c(55, 130))
points(x = stair_height,
    y = heart_rate_2,
    col = "green",
    type = "b",
    lwd = 2)
points(x = stair_height,
    y = heart_rate_3,
    col = "red",
    type = "b",
    lwd = 2)
points(x = stair_height,
    y = heart_rate_4,
    col = "red",
    type = "b",
    lwd = 2)
```



```
time <- c(0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 21)
chick_1 <- c(42, 51, 59, 64, 76, 93, 106, 125, 149, 171, 199, 205)
chick_2 <- c(40, 49, 58, 72, 84, 103, 122, 138, 162, 187, 209, 215)
chick_3 <- c(42, 53, 62, 73, 85, 102, 123, 138, 170, 204, 235, 256)
chick_4 <- c(41, 49, 61, 74, 98, 109, 128, 154, 192, 232, 280, 290)

plot(x = time, y = chick_1,
         type = "l",
         lwd = 2,
         col = "blue",
         ylim = c(40, 300))
points(x = time, y = chick_2,
         type = "l",
         lwd = 2,
         lty = 3,
         col = "blue")
points(x = time, y = chick_3,
         type = "l",
         lwd = 2,
         lty = 1,
         col = "red")
points(x = time, y = chick_4,
         type = "l",
         lwd = 2,
         lty = 3,
         col = "red")
```



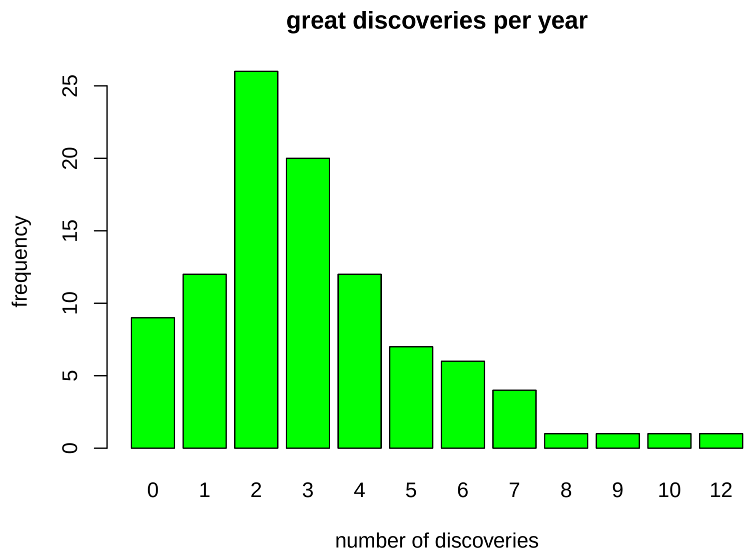
```r
maxima <- c(max(chick_1), max(chick_2), max(chick_3), max(chick_4))

barplot(maxima,
    names = c("Chick 1","Chick 2","Chick 3","Chick 4"),
    ylab = "Maximum weight (grams)",
    col = "gold",
    main = "Maximum chick weights")
```



10.1.7 Discoveries

A

```r
barplot(table(discoveries),
    main = "great discoveries per year",
    xlab = "number of discoveries",
    ylab = "frequency",
    col = "green")
```

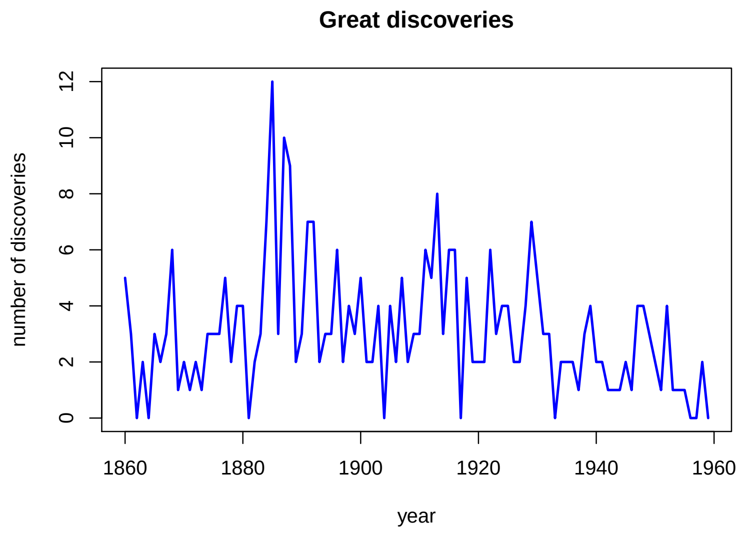


B

```
summary(discoveries)
```

C

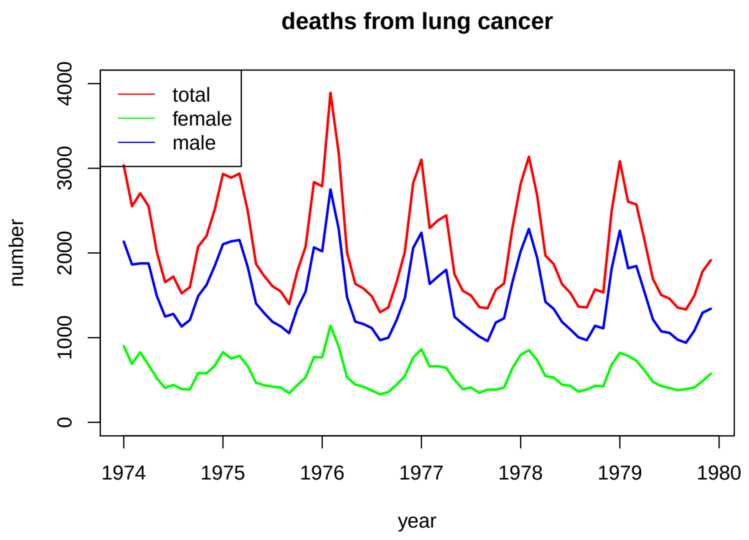
```
plot(discoveries,
         xlab = "year",
         ylab = "number of discoveries",
         main = "Great discoveries",
         col = "blue",
         lwd = 2)
```



10.1.8 Lung cancer

A

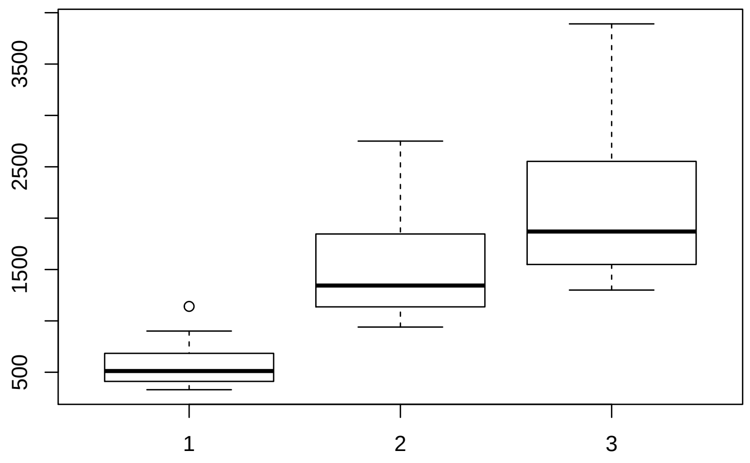
```
total.col <- "red"
m.col <- "blue"
f.col <- "green"
plot(ldeaths,
         main = "deaths from lung cancer",
         xlab = "year",
         ylab = "number",
         col = total.col,
         ylim = c(0, 4000),
         lwd = 2
)
lines(fdeaths, col = f.col, lwd = 2)
lines(mdeaths, col = m.col, lwd = 2)
legend(
    "topleft",
    legend = c("total", "female", "male"),
    col = c(total.col, f.col, m.col),
    lty = 1)
```



B

Create a combined boxplot of the three time-series. Are they indicative of a normal distribution? Are there outliers? If so, can you figure out when this occurred?

```
boxplot(
    fdeaths, mdeaths, ldeaths
)
```



10.2 Complex datatypes

10.2.1 Creating factors

A

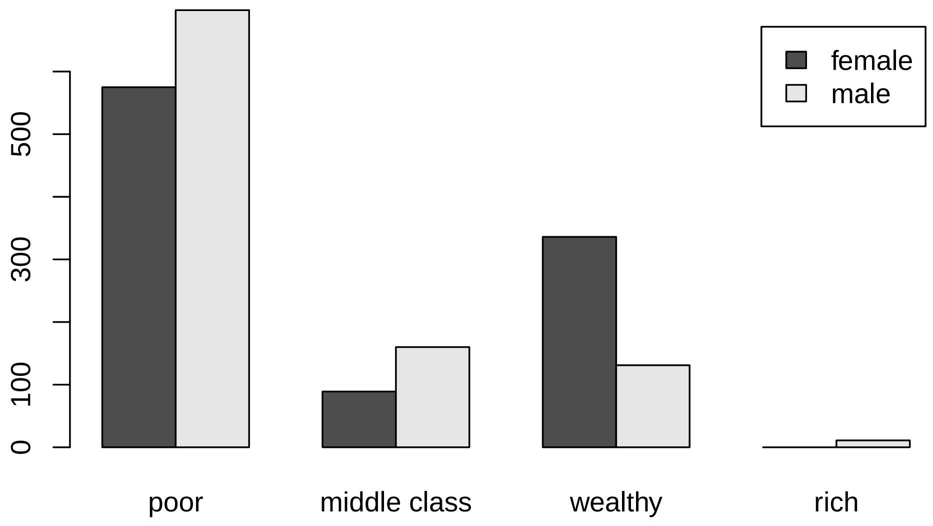
```
animal_risk <- c(2, 4, 1, 1, 2, 4, 1, 4, 1, 1, 2, 1)
animal_risk_factor <- factor(x = animal_risk,
                             levels = c(1, 2, 3, 4),
                             labels = c("harmless", "risky", "dangerous", "deadly"),
                             ordered = TRUE)
barplot(table(animal_risk_factor))
```

B

```
set.seed(1234)
wealth_male <- sample(x = letters[1:4],
                 size = 1000,
                 replace= TRUE,
                 prob = c(0.7, 0.17, 0.12, 0.01))
wealth_female <- sample(x = letters[1:4],
                 size = 1000,
                 replace= TRUE,
                 prob = c(0.8, 0.15, 0.497, 0.003))

wealth_labels <- c("poor", "middle class", "wealthy", "rich")

wealth_male_f <- factor(x = wealth_male,
                        levels = letters[1:4],
                        labels = wealth_labels,
                        ordered = TRUE)

wealth_female_f <- factor(x = wealth_female,
                        levels = letters[1:4],
                        labels = wealth_labels,
                        ordered = TRUE)

#combine
wealth_all_f <- factor(c(wealth_male_f, wealth_female_f),
                        levels = 1:4,
                        labels = wealth_labels,
                        ordered = TRUE)

prop.table(table(wealth_all_f)) * 100
```

```
## wealth_all_f
##         poor middle class      wealthy         rich
##        63.65        12.45        23.35         0.55
```

```
#getting this data right may be a bit of a challenge...
bar_data <- rbind(table(wealth_female_f), table(wealth_male_f))
rownames(bar_data) <- c("female", "male")

barplot(bar_data, beside = T, legend = rownames(bar_data))
```



10.2.2 A dictionary with a named vector

A

```r
codons <- c("G", "P", "K", "S")
names(codons) <- c("GGA", "CCU", "AAA", "AGU")

my_DNA <- "GGACCUAAAAGU"
my_prot <- ""
for (i in seq(from = 1, to = nchar(my_DNA), by = 3)) {
        codon <- substr(my_DNA, i, i+2)
        my_prot <- paste0(my_prot, codons[codon])
}
print(my_prot)
```

```
## [1] "GPKS"
```

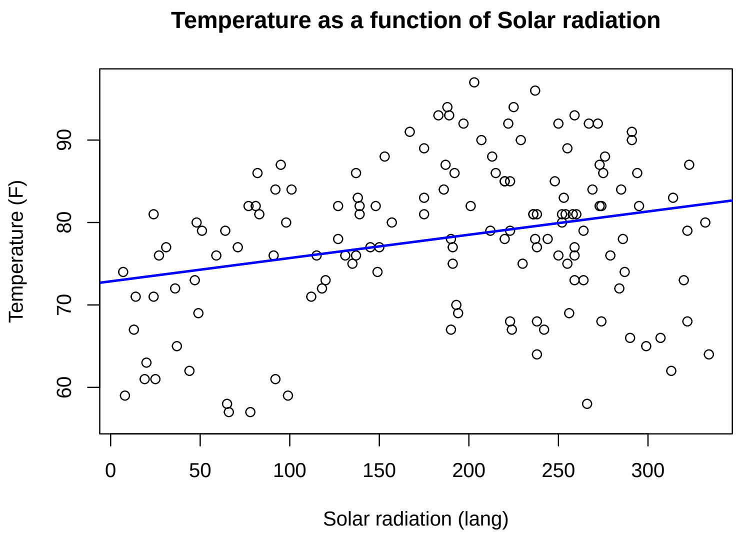
B

```r
nuc_weights <- c(491.2, 467.2, 507.2, 482.2)
names(nuc_weights) <- c('A', 'C', 'G', 'U')

mol_weight <- 0
for (i in 1:nchar(my_DNA)) {
        nuc <- substr(my_DNA, i, i);
        print(nuc)
        mol_weight <- mol_weight + nuc_weights[nuc]
}
mol_weight
```

10.2.3 airquality

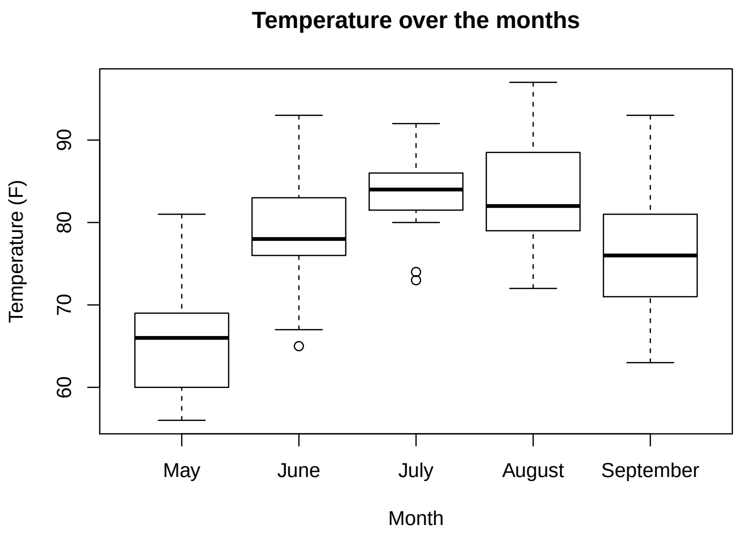
A

```r
plot(airquality$Solar.R, airquality$Temp,
         main = "Temperature as a function of Solar radiation",
         xlab = "Solar radiation (lang)",
         ylab = "Temperature (F)")
abline(lm(airquality$Temp ~ airquality$Solar.R), col = "blue", lwd = 2)
```



B

```r
with(airquality,
        boxplot(Temp ~ Month,
        main = "Temperature over the months",
        xlab = "Month",
        ylab = "Temperature (F)"))
```



C

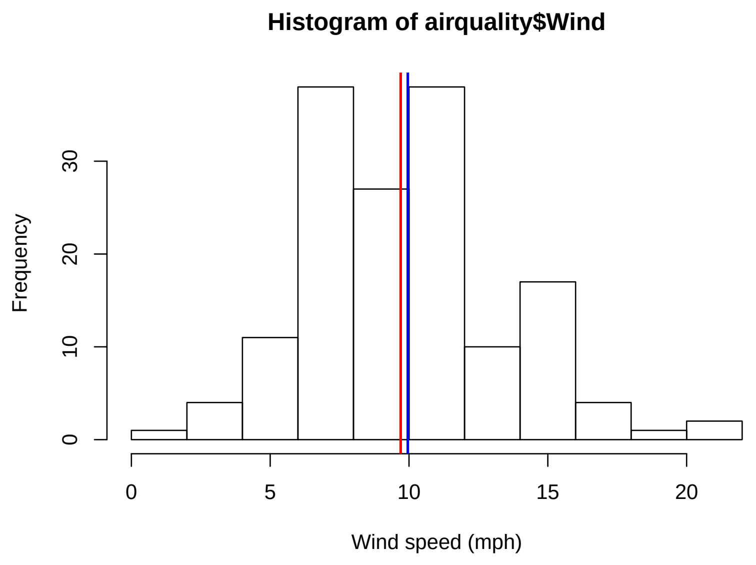
```r
#first create Temp Celcius column:
#(°F    -    32)    x    5/9 = °C
airquality$Temp.C <- (airquality$Temp - 32) * 5/9
```

```
#get the required data
airquality[airquality$Temp.C == min(airquality$Temp.C), c("Temp.C", "Month", "Day")]
```

```
##      Temp.C Month Day
## 5 13.33333   May   5
```

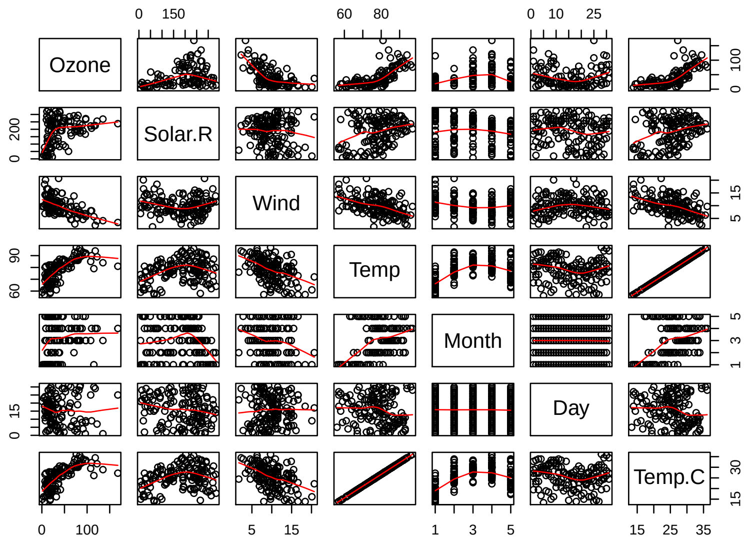
D

```
hist(airquality$Wind, xlab = "Wind speed (mph)")
abline(v = mean(airquality$Wind), col = "blue", lwd = 2)
abline(v = median(airquality$Wind), col = "red", lwd = 2)
```



E

```
pairs(airquality, panel = panel.smooth)
```



Calculate pairwise correlation.

```r
cor(na.omit(airquality))
```

10.2.4 Bird observations

```r
bird_obs <- read.table("data/Observations-Data-2014.csv",
                                                        sep=";",
                                                        head=T,
                                                        na.strings = "",
                                                        quote = "",
                                                        comment.char = "")
```

A

```r
## look at the loaded data structure
str(bird_obs)
```

Apparently, all variables are loaded as a factor; also the `Date.start`, `Date.end` (should be dates of course), `Number` (should be `integer`) and `Notes` (should be `character`) columns. In the original column names there are spaces and these are replaced by dots. First column `Species..` is a serial number and the second `Species` is the English species name.

B

```r
nrow(bird_obs)
```

C

```
class(bird_obs$Number)
```

D

```
bird_obs$Count <- as.integer(bird_obs$Number)
head(bird_obs[, c(4, 8, 14)], n=50)
```

```
##                    Common.name Number Count
## 1  Greater White-fronted Goose      1     1
## 2  Greater White-fronted Goose      6    58
## 3  Greater White-fronted Goose      1     1
## 4  Greater White-fronted Goose      1     1
## 5  Greater White-fronted Goose      2    22
## 6                   Snow Goose      1     1
## 7                 Ross's Goose      1     1
## 8                 Ross's Goose      1     1
## 9                 Ross's Goose      1     1
## 10                Ross's Goose      1     1
## 11                       Brant    3-6    41
## 12                       Brant      1     1
## 13                       Brant    300    43
## 14                       Brant      1     1
## 15                       Brant      3    36
## 16                       Brant      2    22
## 17                       Brant      9    68
## 18              Cackling Goose      3    36
## 19              Cackling Goose      1     1
## 20              Cackling Goose      1     1
## 21              Cackling Goose      1     1
## 22              Cackling Goose      1     1
## 23              Cackling Goose      3    36
## 24              Trumpeter Swan      6    58
## 25                 Tundra Swan      2    22
## 26                 Tundra Swan      1     1
## 27                 Tundra Swan      2    22
## 28                 Tundra Swan      3    36
## 29                 Tundra Swan      2    22
## 30                 Tundra Swan      1     1
## 31                 Tundra Swan      3    36
## 32                 Tundra Swan      1     1
## 33                 Tundra Swan    145    16
## 34                 Tundra Swan      6    58
## 35                 Tundra Swan     18    21
## 36                 Tundra Swan      3    36
## 37                   Wood Duck      1     1
## 38                     Gadwall      2    22
## 39                     Gadwall      3    36
## 40                     Gadwall      1     1
## 41             Eurasian Wigeon      1     1
## 42             American Wigeon      2    22
## 43             American Wigeon      3    36
## 44             American Wigeon      1     1
## 45             American Wigeon      1     1
## 46             American Wigeon    1-2     2
## 47             American Wigeon    2-5    27
## 48            Blue-winged Teal      3    36
## 49            Blue-winged Teal      1     1
## 50            Blue-winged Teal      1     1
```

The factor ***levels*** have been converted into integers, not the original values!

E

```
#read with as.is argument
bird_obs <- read.table("data/Observations-Data-2014.csv",
                                sep=";",
                                head=T,
                                na.strings = "",
                                quote = "",
                                comment.char = "",
                                as.is = c(1, 6, 7, 8, 13))
str(bird_obs)
```

```
## 'data.frame':    2019 obs. of  13 variables:
##  $ Species..  : chr  "4" "4" "4" "4" ...
##  $ Genus      : Factor w/ 166 levels "Accipiter","Agelaius",..: 8 8 8 8 8 38 38 38 38 38 ...
##  $ Species    : Factor w/ 300 levels "aalge","acuta",..: 11 11 11 11 11 42 235 235 235 235 ...
##  $ Common.name: Factor w/ 329 levels "Acorn Woodpecker",..: 121 121 121 121 121 266 239 239 23
##  $ CBRC.Review: Factor w/ 3 levels "FALSE","N","Y": 2 2 2 2 2 2 2 2 2 2 ...
##  $ Date.start : chr  "3-Jun-14" "28-Jul-14" "1-Sep-14" "2-Sep-14" ...
##  $ Date.end   : chr  "19-Jun-14" NA NA NA ...
##  $ Number     : chr  "1" "6" "1" "1" ...
##  $ Location   : Factor w/ 980 levels " Coyote Creek Trail San Jose",..: 629 639 169 503 28 673
##  $ County     : Factor w/ 9 levels "Alameda","Contra Costa",..: 7 4 9 9 3 9 9 9 4 4 ...
##  $ Observer.1 : Factor w/ 692 levels "A Sojourner",..: 216 351 544 623 333 623 623 623 323 206
##  $ Other.Obs  : Factor w/ 157 levels "Aaron Maizlish",..: NA NA NA NA NA NA NA NA 155 NA ...
##  $ Notes      : chr  "Adult bird seen on golf course grounds with Canada geese!" "Saw 6 along
```

Convert Number column to Count of integers.

```
bird_obs$Count <- as.integer(bird_obs$Number)
```

```
## Warning: NAs introduced by coercion
```

Note that there are other ways to achieve this, e.g. the `colClasses` argument to `read.table()`.

F

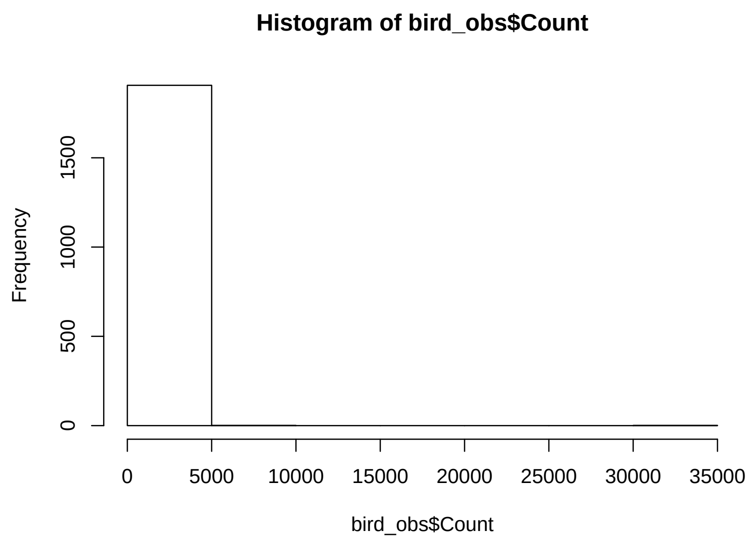
```r
head(bird_obs[, c(4, 8, 14)], n=50)
sum(is.na(bird_obs$Count))
```

G

```r
#What is the maximum number of birds in a single sighting?
bird_obs[which(bird_obs$Count == max(bird_obs$Count, na.rm = T)), ]
##OR
bird_obs[!is.na(bird_obs$Count) & bird_obs$Count == max(bird_obs$Count, na.rm = T), ]

#What is the mean sighting count
mean(bird_obs$Count, na.rm = T)

#What is the median of the sighting count
median(bird_obs$Count, na.rm = T)
```

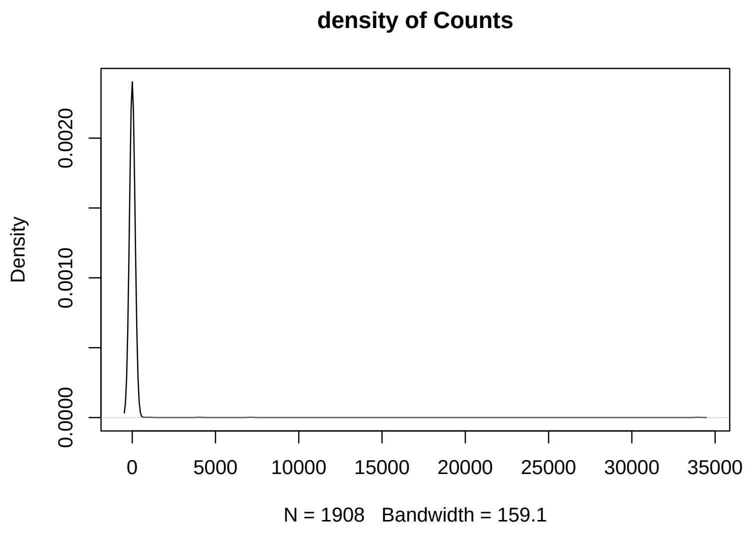
H

```r
hist(bird_obs$Count)
```



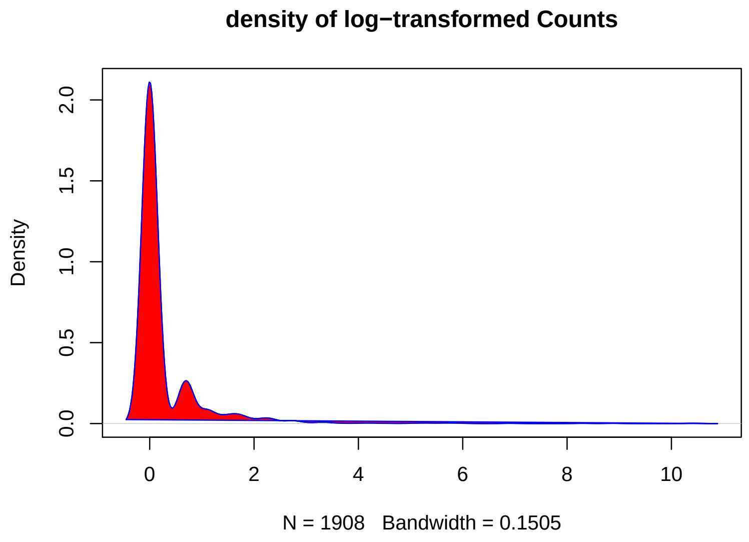
Not very helpful, now is it? Try fiddling with the `breaks` argument.

```r
plot(density(bird_obs$Count, na.rm=T),
         main = "density of Counts")
```



Better results with a log transformation (and some coloring)

```r
d <- density(log(bird_obs$Count), na.rm=T)
plot(d, main = "density of log-transformed Counts")
polygon(d, col = "red", border = "blue")
```



I

```r
#How many different species were recorded?
length(unique(bird_obs$Common.name))

#How many genera do they constitute?
length(unique(bird_obs$Genus))
```

```r
#What species from the genus "Puffinus" have been observed?
#the actual sightings
bird_obs[bird_obs$Genus == "Puffinus", c(2, 3, 4, 6, 14)]
#the species
unique(bird_obs[bird_obs$Genus == "Puffinus", "Common.name"])
```

J

```r
#these are the values that need to be rescued:
table(bird_obs[is.na(bird_obs$Count), "Number"])
#I suggest you take the lowest of the range-like values:
#1-3 becomes 1; 2-3 becomes 2; 100s becomes 100 etc
#then do something like
tmp <- bird_obs$Number[1:50]
tmp
gsub("(\\d+)-(\\d+)", "\\1", tmp)
```

10.3 Regular Expressions

10.3.1 Restriction enzymes

A

```r
pacI_re <- "TTAATTAA"
patterns <- c("T{2}A{2}T{2}A{2}",
           "(TTAA){2}",
           "(T{2}A{2}){2}")
for(ptrn in patterns){
    print(grepl(ptrn, pacI_re))
}
```

B

```r
sfiI_re <- "GGCCACGTAGGCC"
patterns <- c("G{2}C{2}[GATC]{5}G{2}C{2}",
           "GGCC[GATC]{5}GGCC",
           "[GC]{4}[GATC]{5}[GC]{4}") #last one is less specific!
for(ptrn in patterns){
    print(grepl(ptrn, sfiI_re))
}
```

10.3.2 Prosite Patterns

A

PS00211:
“[LIVMFYC]-[SA]-[SAPGLVFYKQH]-G-[DENQMW]-[KRQASPCLIMFW]-

[KRNQSTAVM]-[KRACLVM]-[LIVMFYPAN]-{PHY}-[LIVMFW]-[SAGCLIVP]-{FYWHP}-{KRHP}-[LIVMFYWSTA]."

```
PS00211<- "[LIVMFYC][SA][SAPGLVFYKQH]G[DENQMW][KRQASPCLIMFW][KRNQSTAVM][KRACLVM][LIVMFYPAN][^PHY]
```

B

PS00018: "D-{W}-[DNS]-{ILVFYW}-[DENSTG]-[DNQGHRK]-{GP}-[LIVMC]-[DENQSTAGC]-x(2)- [DE]-[LIVMFYW]."

```
PS00018 <- "D[^W][DNS][^ILVFYW][DENSTG][DNQGHRK][^GP][LIVMC][DENQSTAGC].{2} [DE][LIVMFYW]"
```

10.3.3 Fasta Headers

```
library(stringr)
fasta_headers <- readLines("./data/fasta_headers.txt")
```

A

```
str_match(fasta_headers, "\\[(.+)\\]")[, 2]
str_match(fasta_headers, "\\[([[:alpha:]]+ [[:alpha:]]+) ?(.+)?\\]")[, 2]
```

B

```
str_match(fasta_headers, ">([[:alpha:]]{2,3}\\|\\w+)\\|")[, 2]
```

C

```
str_match(fasta_headers, ">.+\\| (.+?) \\[")[, 2]
```

10.4 Scripting

10.4.1 Illegal reproductions

The mean

```
my_mean <- function(x) {
        sum(x, na.rm = T) / length(x)
}
```

Standard deviation

```
my_sd <- function(x) {
        sqrt(sum((x - mean(x))^2)/(length(x)-1))
}
```

Median

```
my_median <- function(x) {
        sorted <- sort(x)
        if(length(x) %% 2 == 1) {
                #uneven length
                my_median <- sorted[ceiling(length(x)/2)]
        } else {
                my_median <- (sorted[length(x)/2] + sorted[(length(x)/2)+1]) / 2
        }
        return(my_median)
}
```

10.4.2 Interquantile ranges

```
interquantile_range <- function(x, lower = 0, upper = 1) {
  if (! is.numeric(x) |
      ! is.numeric(lower) |
      ! is.numeric(upper)) {
    stop("all three arguments should be numeric")
  }
  lower_val <- quantile(x, probs = lower)
  upper_val <- quantile(x, probs = upper)
  tmp <- upper_val - lower_val
  #a named vector is always nice, for acces but also for display purposes
  names(tmp) <- paste0(lower*100, "-", upper*100, "%")
  tmp
}
tst <- rnorm(1000)
interquantile_range(tst) # 0 to 1
interquantile_range(tst, 0.25, 0.75) # custom
#interquantile_range("foo") # error!
```

Perform some tests on the arguments to make a robust method: are all arguments numeric?

To test you method, you can compare `interquantile_range(some_vector, 0.25, 0.75)` with `IQR(some_vector)` - they should be the same.

10.4.3 Vector distance

```
distance <- function(p, q) {
    if (! is.numeric(p) | ! is.numeric(q)) {
        stop("non-numeric vectors passed")
    }
    if (length(p) != length(q)) {
        stop("vectors have unequal length")
    }
    sqrt(sum((p - q)^2))
}
```

Other distance measures

`[NO SOLUTION YET]`

10.4.4 G/C percentage of DNA

```
GC_perc <- function(seq, strict = TRUE) {
        if (is.na(seq)) {
                return(NA)
        }
        if (length(seq) == 0) {
                return(0)
        }
        seq.split <- strsplit(seq, "")[[1]]
        gc.count <- 0
        anom.count <- 0
        for (n in seq.split) {
                if (length(grep("[GATUCgatuc]", n)) > 0) {
                        if (n == "G" || n == "C") {
                                gc.count <- gc.count + 1
                        }
                } else {
                        if (strict) {
                                stop(paste("Illegal character", n))
                        } else {
                                anom.count <- anom.count + 1
                        }
                }
        }
        ##return perc
        ##print(gc.count)
        if (anom.count > 0) {
                anom.perc <- anom.count / nchar(seq) * 100
                warning(paste("Non-DNA characters have percentage of", anom.perc))
        }
        return(gc.count / nchar(seq) * 100)
}
```

10.5 Function apply and its relatives

```
whale_sel_url <- "https://raw.githubusercontent.com/MichielNoback/davur1/gh-pages/exerc
whale_selenium <- read.table(whale_sel_url,
        header = T,
        row.names = 1)
```

A

```
apply(X = whale_selenium, MARGIN = 2, FUN = mean)
```

B

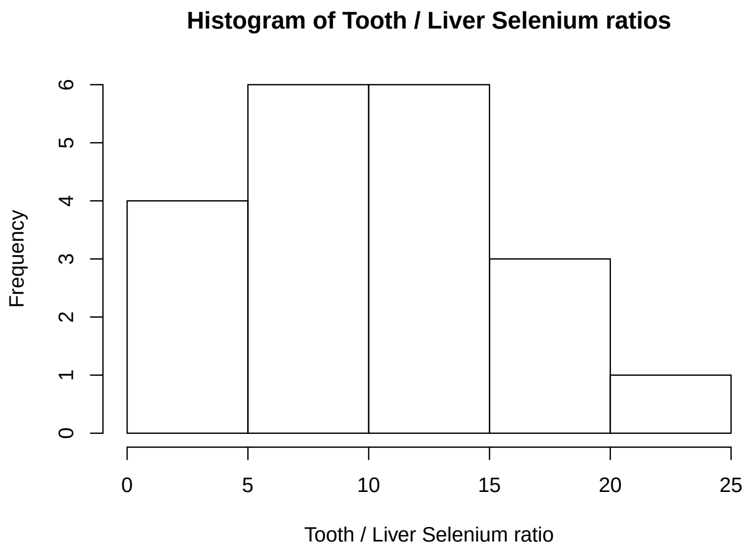
```
apply(X = whale_selenium, MARGIN = 2, FUN = sd)
```

C

```
my.sem <- function(x) {
        sem <- sd(x) / sqrt(length(x))
}
apply(X = whale_selenium, MARGIN = 2, FUN = my.sem)
```

D

```
whale_selenium$ratio <- apply(X = whale_selenium,
            MARGIN = 1,
            FUN = function(x){
                    x[2] / x[1]
            })
hist(whale_selenium$ratio,
         xlab = "Tooth / Liver Selenium ratio",
         main = "Histogram of Tooth / Liver Selenium ratios")
```



E

Inline expressions are like this: 15.4 MpH.

10.5.2 ChickWeight

This exercise revolves around the `ChickWeight` dataset of the built-in `datasets` package.

A

```
#MANY WAYS TO GET THERE
length(split(ChickWeight, ChickWeight$Chick))
```

```
## [1] 50
```

```
#OR
sum(tapply(ChickWeight$Diet, ChickWeight$Chick, FUN = function(x){1}))
```

```
## [1] 50
```

```
#OR
length(unique(ChickWeight$Chick))
```

```
## [1] 50
```

```
#OR
nrow(aggregate(x = ChickWeight, by = list(ChickWeight$Chick), FUN = function(x){x}))
```

```
## [1] 50
```

B

```
aggregate(formula = weight ~ Diet, data=ChickWeight, FUN = mean, na.rm = T)
```

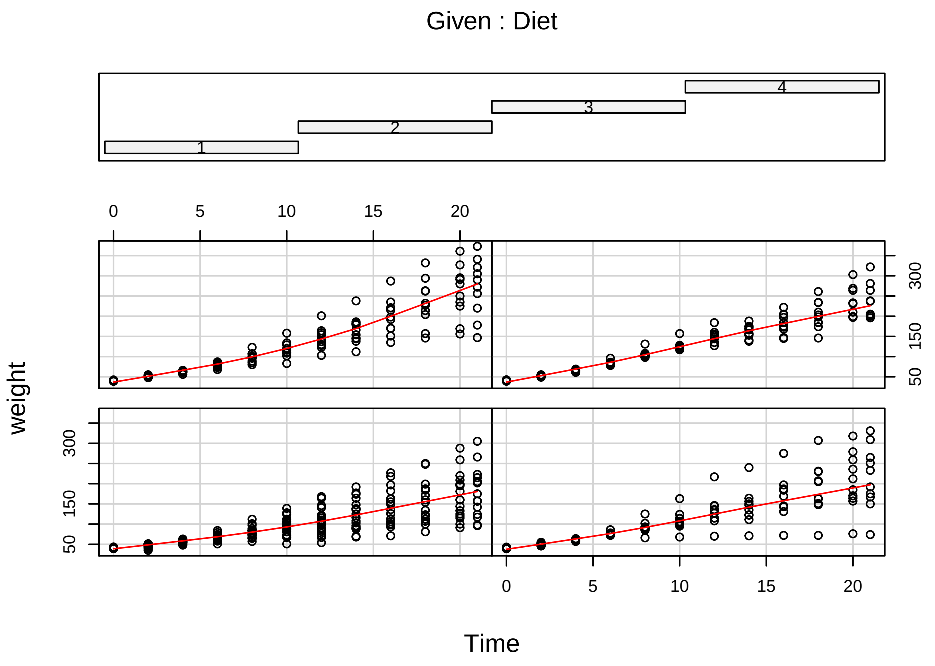
```
##   Diet   weight
## 1    1 102.6455
## 2    2 122.6167
## 3    3 142.9500
## 4    4 135.2627
```

```
#OR
aggregate(x = ChickWeight$weight, by = list(Diet = ChickWeight$Diet), FUN = mean, na.r
```

```
##   Diet        x
## 1    1 102.6455
## 2    2 122.6167
## 3    3 142.9500
## 4    4 135.2627
```

C

```
coplot(weight ~ Time | Diet, data = ChickWeight, panel = panel.smooth)
```



D

```
#A naive for-loop here - is this the best solution?
ChickWeight$weight_gain <- NA #create the column with missing values
for (i in 1:nrow(ChickWeight)) {
        #skip first row and rows that are preceded by values for another chick
        if (i > 1 && ChickWeight$Chick[i] == ChickWeight$Chick[i-1]) {
                ChickWeight[i, "weight_gain"] <- ChickWeight$weight[i] - ChickWeight$weight[i-1]
        }
}
```

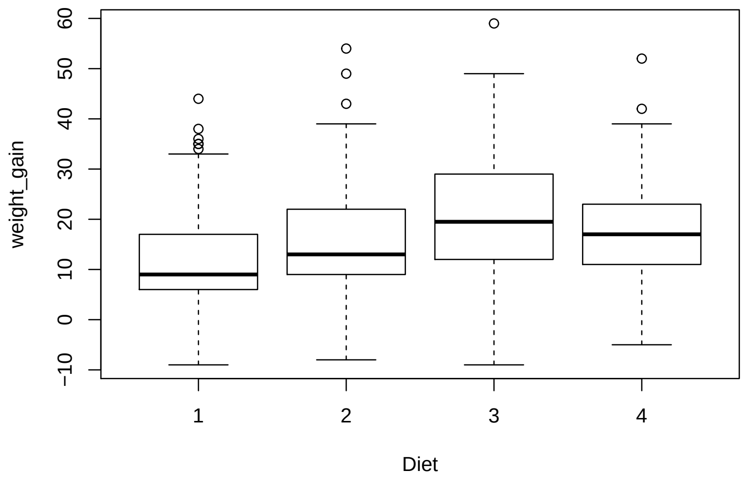
E

```
local_file <- "ChickWeight_weight_gain.Rdata"
download.file(paste0("https://github.com/MichielNoback/davur1_gitbook/raw/master/data/", local_fi
load(local_file)
#attach
ChickWeight$weight_gain <- stored.weight.gain
```

```
tapply(X = ChickWeight$weight_gain, INDEX = ChickWeight$Diet, FUN = mean, na.rm = T)
#or with aggregate
aggregate(formula = weight_gain ~ Diet, data = ChickWeight, FUN = median)
#or with split and sapply
sapply(split(ChickWeight[, "weight_gain"], ChickWeight$Diet), sd, na.rm = T)
```

F

```
boxplot(weight_gain ~ Diet, data = ChickWeight)
```



10.5.3 Food constituents

A

```
foods <- read.table(
        "https://raw.githubusercontent.com/MichielNoback/davur1_gitbook/master/data/food_constitu
```

```
levels(foods$Type)
table(foods$Type)
```

B

```
mean(foods[foods$Type == "chocolate", "kcal"])
```

C

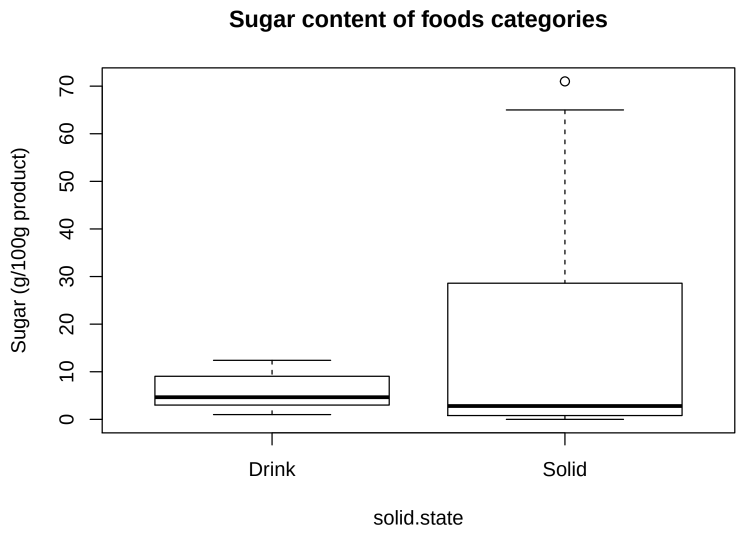
```
#aggregate over Type
mean.fat <- aggregate(formula = fat.total ~ Type, data = foods, FUN = mean)
#order and select first
mean.fat[order(mean.fat$fat.total, decreasing = T)[1], ]
```

D

```
mean.energy <- aggregate(formula = kcal ~ Type, data = foods, FUN = mean)
mean.energy[order(mean.energy$kcal)[1], ]
mean.energy[order(mean.energy$kcal, decreasing = T)[1], ]
```

E

```
#more verbose means possible; this efficient way demonstrating use of %in%
foods$solid.state <- !foods$Type %in% c("milk", "beverage")
boxplot(formula = carb.sugar ~ solid.state,
                data = foods,
                main = "Sugar content of foods categories",
                names = (c("Drink", "Solid")),
                ylab = "Sugar (g/100g product)")
```



F

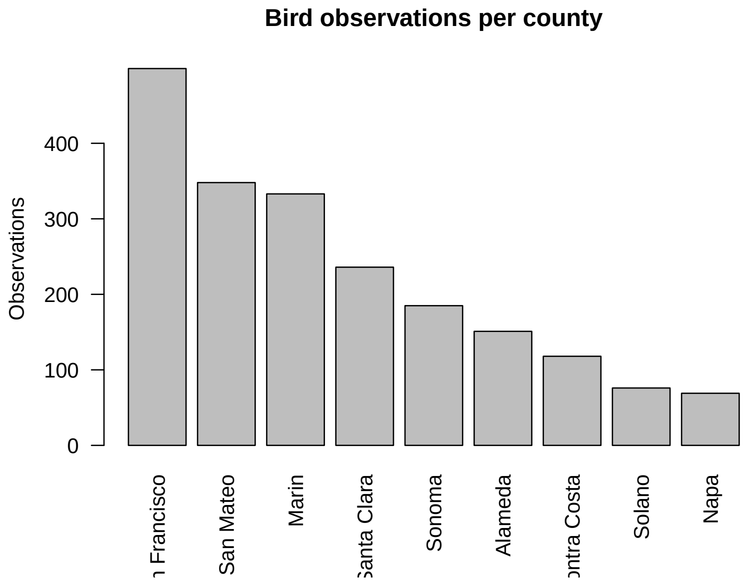
NOWORKEDSOLUTIONHERE

10.5.4 Bird observations revisited

```
bird_obs <- read.table("data/Observations-Data-2014.csv",
                                sep=";",
                                head=T,
                                na.strings = "",
                                quote = "",
                                comment.char = "",
                                as.is = c(1, 6, 7, 8, 13))
bird_obs$Count <- as.integer(bird_obs$Number)
```

A

```
c.split <- split(x = bird_obs, f = bird_obs$County)
c.counts <- sapply(c.split, nrow)
barplot(c.counts[order(c.counts, decreasing = T)],
                main = "Bird observations per county",
                ylab = "Observations",
                las = 2)
```



B

```
obs.split <- split(x = bird_obs, f = bird_obs$Observer.1)
obs.counts <- sapply(obs.split, nrow)
obs.counts <- obs.counts[obs.counts > 10]
obs.counts[order(obs.counts, decreasing = T)]
```

C

```
obs.counts[order(obs.counts, decreasing = T)][1]
```

D

```
g.split <- split(bird_obs, bird_obs$Genus)
g.species <- lapply(g.split, function(x) {
        unique(x$Common.name)
})
#create ordering
g.species.count <- sapply(g.species, length)
g.order <- order(g.species.count, decreasing = T)
#apply order to list and select only first five
g.species[g.order[1:5]]
```

E

```
bird_obs$Date.start <- as.Date(bird_obs$Date.start, format = "%d-%b-%y")
date.series <- aggregate(Count ~ Date.start, data = bird_obs, FUN = sum, na.rm = T)
#2024 is an error input, remove it
date.series <- date.series[1:nrow(date.series)-1, ]
plot(x = date.series$Date.start, y = date.series$Count, ylim = c(0, 250))
```