Learning Data Science in R

Antonio Fidalgo

Last update: March 25, 2019

Contents

Foreword

> "Data science is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it…"
>
> — paraphrase of Dan Ariely's quote on big data

There is already a large number of excellent and free references for learning data science in R. The list would be too vast, but two names stand out: Hadley Wickham and Yihui Xie.
Work of the first includes the books Wickham and Grolemund (2016) and Wickham (2014) (freely available, including source code, on github.com/hadley) as well as some of the most popular packages in R such as `ggplot2` (Wickham et al., 2018) and `tidyverse`(Wickham, 2017).
Work of the second is used precisely in the current file thanks to the `rmarkdown` package (Allaire et al., 2018), `knitr` (Xie, 2018b) and `bookdown` (Xie, 2018a). His source code is also on github.com/yihui/.

So, the question arises of why one should even bother to write on the topic. The present "book" is justified on the following grounds.

1. There is no better way of learning data science and R than **doing oneself** the pages of code.

2. The material gathered here is a **personal selection** made on what I judge most relevant for my work, the most important techniques in general or, sometimes, the least obvious for the regular practitioner.

3. These notes serve as an **archive** of what I did in the domain so that I can easily access it in the future.

Part I

Getting started

Chapter 1

Getting set

1.1 Install applications

Download the following free applications, available on all platforms:

1. R
2. RStudio, free Desktop version
3. A Latex distribution (e.g., MacTex for Mac or MiKTeX for Windows machines)

The first two are easily and quickly installed. The last is a very large program (a few Gb) and needs time to install.

Another application needed is a

4. Git distribution

This is also a free software.
Once you have installed Git for version control, activate it in RStudio: *Tools> Global Options> Git/SVN* and click on *Enable version control interface for RStudio projects.*
Also generate a SHH RSA key. We will use it to identify at the GitHub repo.

1.2 Sign for GitHub

Create an account at GitHub at https://github.com.

1.3 Create your project

File> New project> Existing Directory and chose that book folder.
Now, every time you create content for your book, you must start a Rstudio session *File> Open Project...*
All the files of the project are the files of the folder, and vice versa.

1.4 Create GitHub repo and link your machine to it

Create a new repository (pronounced ‘repo’) whose name is **exactly** the same as your R project / book folder (e.g., ‘myRbook’).

On the top left menu in GitHub, go to *Settings> SHH and GPG keys* and click on 'New SHH key'. Paste the SHH key generated by RStudio.

In RStudio go to *Tools> Project Options...> Git/SVN.* Under *Version control system*, select 'Git'.
Still in RStudio, *Tools> Terminal> New Terminal.* This open a Terminal where you type commands.

Start by initializing Git on the terminal.

```
git init
```

Then paste the message shown at the creation of the repo (changing the names, of course):

```
git remote add origin https://github.com/YOURNAME/YOURREPO.git
git push -u origin master
```

Your local `master` should now be connected to the `master` on GitHub.
If necessary, restart RStudio. At the restart, a Git thumbnail should appear in a pane. You are ready to commit and push your files.

1.4.1 Troubleshooting

If the procedure above works, great! One must be aware, however, that it sometimes hits some problem unexpected and difficult to understand problem. In these cases, a classic solution is to copy and google the error message with a first term such as "RStudio".
In my experience, the following hack worked. Find the invisible file of the repo *.git> config*. Hidden files on Mac are turned visible by the keyboard key `CMD + SHIFT + .`. In that file, delete the following part (including the space it takes)

```
[remote "origin"]
    url = git@github.com:USERNAME/repo.git
    fetch = +refs/heads/*:refs/remotes/origin/*
```

Then, try again the procedure above.

1.5 Collaboration on GitHub

There are various established ways for collaborating on a GitHub repo. As a matter of facts, collaboration on a project is the *raison d'etre* of GitHub.
We illustrate here the easiest of them, namely the joint reading and writing into one repo by all the members of a team.
Importantly, it is assumed that all members of the group are able to push/pull code from RStudio to an experimental repo on GitHub (see above).
The following are the steps in the collaboration's setup.

1.5.1 Owner of the organization

One member of the group creates an organization on GitHub under *Settings> Organizations> New organization.*
Then, under that organization, this member of the team creates a new repo whose name is the same as the folder and R project that the group will work on (e.g., 'ouRbook').
The owner of the organization connects from RStudio to the repo and populates it with the current files of the project.
Still as a task of the owner, create a team. Under the organization page in GitHub, simply click on 'New team' and give it a name.
A team now appears in the organization page. In the page of that team, click on 'Add a member'. A window

appears where the member to be added can be searched for and added.
Once that member is found and selected, make the double step of adding him/her to the team AND assign him/her to the repo. This second step can still be made later, but it's simpler to do it right then.
Importantly, make sure that the new member has reading and **writing** rights into the repo. This again, can be changed later by navigating the team's page.

1.5.2 Member of the team

After the steps above, the newly added member receives an email to confirm the participation the organization / team.
Navigate now to the team's page and locate the repo that the owner has created.
A button-menu allows to 'Clone or download the repo'. Click to show the https address of the repo, `https://github.com/ORGANIZATION/repo.git` and copy that link.
Open a simple RStudio session (not a project!). Open a new Terminal (*Tools> Terminal> New Terminal*) and type `cd` (change directory) and a path to the folder where you want the repo to be saved. For instance,

```
cd Dropbox/Fresenius/DS4B
```

The terminal is writing into that folder, as indicated by the path before the `$` sign in command line of the terminal.
Recall that you copied the address of the team's repo. Then, in command line, now type the following commands and your copied address:

```
git clone https://github.com/ORGANIZATION/repo.git
```

The whole repo is now a new folder in your directory. If all went well, you can now pull and push into that repo in GitHub.

Part II

Source file and output files

Chapter 2

Rmd files

This chapter gathers general comments about `.Rmd` files.

2.1 Options for all chuncks

It is convenient to set options for all the R chuncks of the document. This saves time when writing these chuncks.
A natural place to set these options is in a first R chunck.

```
knitr::opts_chunk$set(OPTION1 = TRUE/FALSE,
                      OPTION2 = TRUE/FALSE,
                      ...)
```

Importantly, these options are overriden by the particular chunck options.

~~~
```{r, OPTION2=FALSE}
~~~

Options actually take R code. So, the following are examples that could be used to define the option.

~~~
```{r, eval=4>3, echo=format(Sys.Date(), '%Y-%B-%d') > '2019-March-10'}
# eval is always TRUE
# echo = TRUE if current date is after March 10, 2019
```
~~~

The list of options can be found here https://yihui.name/knitr/options/. Below are some comments on some of these options (the least trivial for the author).

- `collapse` determines whether the source code and the ouput should be merged into a single block. Here is the same chunck with different values of the option:

`collapse=TRUE`

```
2+ 2
#> [1] 4
3* 5
#> [1] 15
```

`collapse=FALSE`

```
2+ 2
```

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

```
3* 5
```

```
#> [1] 15
```

- `comment` gives the string to be printed before the output.

```
comment='##'
```

```
2+ 2
## [1] 4
```

```
comment='R>'
```

```
2+ 2
R> [1] 4
```

Worth noting: a `#` as a first character of the comment string (with `collpase=TRUE`) turns the output font into a comment-like text.

- `child` allows a document to call and use another file as input in the document.

~~~
```{r, child='PATH/TO/OTHER/file.Rmd'}
~~~

The path can be either absolute or relative.
For relative paths, the following applies:

- `~/` starts a path a the root,
- `../` indicates the parent directory,
- `../../` for parent of the parent directory,
- to move forward, start with the name of the included folder in the current directory.

2.2 Latex code

The overwhelming reason to introduce Latex code in a `.Rmd` file is for typesetting mathematical expressions. There are two main ways to type math in Latex:

- in the text, surrounded by special delimiters, `\( math \)` (alternatively, one can use the deprecated `$ math $`),
- in an equation, surrounded by special delimiters, `\[ math equation \]`, or in a dedicated environment such as `\begin{equation} math equation \end{equation}` (also deprecated, `$$ math equation $$`).

Math format is usually distinct from the text format. For instance, compare a^2 + b^2 = c^2 (simple markdown) to $a^2 + b^2 = c^2$ (Latex). Notice, however, that most mathematical symbols are not supported by markdown. This why basic knowledge of Latex is essential for producing documents with math formula.
The first part of that knowledge consists in the list of Latex symbols. These are typed into a document starting with a slash `\`. For instance, the Latex symbol α is produced by the code `\alpha` while $\sum$ is given by `\sum`.
Superscripts and subscripts are produced with the characters `^` and `_`, respectively, and the content of the super- and subscripts enclosed within `{ }`. For instance, x_2^3 is produced by the code `x_{2}^{3}` (or `x^{3}_{2}`) and $\sum_{i=1}^4$ is given by `\sum_{i=1}^{4}`.
A comprehensive list of symbols (some of them requiring special Latex packages) can be found in Pakin (2009). Shorter lists can be found online, including on wiki.
A web-based application helps beginners by offering suggestions to a hand-drawn symbol, detexify.
A second bulk in the knowledge of Latex is the understanding of special environments. But then, if a document requires lots of special Latex environments, then `.Rmd` files are not the most appropriate type of files to use.
Some of these environments, however, have a perfect integration in `.Rmd` files, such as the most common, the equation. Here is an example, given by the code

```
\[
\sum_{i=0}^{\infty} ak^{i} = \frac{a}{1-k} \text{, for }  \lvert k \rvert < 1
\]
```

$$\sum_{i=0}^{\infty} ak^{i} = \frac{a}{1-k}\text{, for } \lvert k \rvert < 1$$

Chapter 3

Customize output

This chapter is about choosing and/or modifying the way the output file looks like.

3.1 Multiple built-in output types

[complete this section] From pdf to slides, through webpages and notebooks.

3.2 New types provided by packages

[complete this section] These are more variations of the above.

3.3 CSS: custom html

A file to change how html outputs look like.

3.4 Latex preamble

This file is added to the preamble of the Latex file to modify how the pdf output is compiled.

Part III

R Basics

Chapter 4

R as a calculator

R can be used as a simple calculator. For instance, 45+17=62 .
Here are a few more examples of the commands.

4.1 Usual operators

4.1.1 Simple operations

The usual symbols `+, -, *, /` apply .

```
2 + 6
#> [1] 8
56/ 6
#> [1] 9.333333
```

4.1.2 Parentheses

The parentheses work as expected. But they are also a common source of error when they are not matched.

```
(4+3)*((7-3)/(1+.05))
#> [1] 26.66667
```

The next expression will generate an error and prevent the compilation of the whole book.

```
(4+3)*((7-3/(1+.05))
```

4.1.3 Exponents

There are two ways of expressing the power of a number: `^` and `**`.

```
3^4
#> [1] 81
3**4
#> [1] 81
```

4.2 Unusual operators

4.2.1 Special operations

The symbols `%/%` and `%%` return the entire part of the result of the division and the rest of the division, respectively.

```
56/6
#> [1] 9.333333
56%/%6
#> [1] 9
56%%6
#> [1] 2
```

4.3 Usual functions

The common functions found in any calculator also have an equivalent on R. The following examples need no further comment.

```
log(100)
#> [1] 4.60517
sqrt(100)
#> [1] 10
```

4.4 Unusual functions

There are several other functions applied to numbers that one does not usually find in a calculator. A few examples below:

```
# floor / ceiling for the closest integer
floor(13.47)
#> [1] 13
ceiling(13.47)
#> [1] 14
```

Chapter 5

Data stuctures

This chapter lists the most common objects to store data.

5.1 Atomic vectors

The basic data structure in R is the vector. Vectors have three characteristics:

- a type, `typeof()`,
- a length, `length()`,
- some attributes, `attributes()`.

5.1.1 Types of data

There are four common types of atomic vectors:

- logical,
- integer,
- double (often called numeric),
- character.

Note that if vector elements not all of the same type, R makes coercion. In that case, the order becomes: "logical < numeric < character".

Here are a few illustrations.

```
# logical vector
log_vector <- c(TRUE, FALSE, FALSE)
log_vector
#> [1]  TRUE FALSE FALSE

# numeric vector
num_vector <- c(12, 10, 3)
typeof(num_vector)
#> [1] "double"

# integer vector
int_vector <- c(12L, 10L, 3L)
typeof(int_vector)
#> [1] "integer"
```

```
# character vector
chr_vector <- c("a", "b", "c")
typeof(chr_vector)
#> [1] "character"

# mixed types vector
mix_vector <- c(2, "a")
is.numeric(mix_vector)
#> [1] FALSE
typeof(mix_vector)
#> [1] "character"
```

Numeric vectors can be created with the shortcut `:`, which is used to imply all the integers from the number on the left of `:` to the number on its right.

```
vector_A <- c(1:5)
vector_A
#> [1] 1 2 3 4 5
length(vector_A)
#> [1] 5
```

5.1.2 Factor vector

This is a special type of vector. It has a limited number of values, called levels. These levels can be unordered (e.g., gender is either "Female" or "Male") or ordered (e.g. school level is "Primary", "Secondary", "Tertiary")

```
gender <- factor(c("Male", "Male", "Female", "Male", "Female", "Female", "Male"))
gender
#> [1] Male   Male   Female Male   Female Female Male
#> Levels: Female Male
levels(gender)
#> [1] "Female" "Male"
summary(gender)
#> Female   Male
#>      3      4
school <- factor(c("Primary", "Secondary", "Tertiary"),
                    ordered=TRUE)
school
#> [1] Primary   Secondary Tertiary
#> Levels: Primary < Secondary < Tertiary

school2 <- factor(c("Primary", "Secondary","Secondary", "Tertiary"),
                    labels=c( "Secondary", "Tertiary","Primary"),
                    ordered=TRUE)
school2
#> [1] Secondary Tertiary  Tertiary  Primary
#> Levels: Secondary < Tertiary < Primary
```

5.2 Matrices and arrays

R is not often used for matrices calculations: it is too slow for that and there are better programs for it out there (e.g. Matlab).

```
# populate a matrix with the elements in of the vector, and give the dimensions
M <- matrix(c(4, 1, 0, 3, 6, 8), nrow=3, ncol=2)
M
#>      [,1] [,2]
#> [1,]    4    3
#> [2,]    1    6
#> [3,]    0    8
```

If we think of a matrix as a 2 dimensions vector, then arrays are n dimensions vectors. Is it important? It might be in some very specific cases.

```
mya <- array(data=1:18, dim=c(2,3,3))
mya
#> , , 1
#>
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
#>
#> , , 2
#>
#>      [,1] [,2] [,3]
#> [1,]    7    9   11
#> [2,]    8   10   12
#>
#> , , 3
#>
#>      [,1] [,2] [,3]
#> [1,]   13   15   17
#> [2,]   14   16   18
```

5.3 Lists

These are the one-size fit all structure… A list is an object composed of any other object, even… another list! Very useful data structure!

```
school<-factor(c("Primary", "Secondary", "Tertiary"), ordered=TRUE)
mylist <- list(numbers=c(1:60),
                somenames=c("Jim","Jules"),
                results= c(T,F,F,T),
                school=school)
mylist
#> $numbers
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
#> [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
#> [47] 47 48 49 50 51 52 53 54 55 56 57 58 59 60
#>
#> $somenames
#> [1] "Jim"   "Jules"
```

```
#>
#> $results
#> [1]  TRUE FALSE FALSE  TRUE
#>
#> $school
#> [1] Primary   Secondary Tertiary
#> Levels: Primary < Secondary < Tertiary

# change the names of the elements of the list
names(mylist) <- c("N", "O","R","S")
```

5.4 Data frames

Data frames are the second most important data structure in R. We can think of it as a better version of a data set in Excel. It stacks together observations over many variables, each of these variables being a vector.

```
# mtcars is a built-in data set
data(mtcars)
class(mtcars)
#> [1] "data.frame"
mtcars
#>                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
#> Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
#> Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
#> Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
#> Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
#> Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
#> Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
#> Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
#> Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
#> Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
#> Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
#> Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
#> Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
#> Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
#> Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
#> Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
#> Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
#> AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
#> Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
#> Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
#> Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
#> Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
#> Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
#> Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
#> Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
#> Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
#> Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
head(mtcars)
#>                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
#> Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
#> Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
#> Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
str(mtcars)
#> 'data.frame':    32 obs. of  11 variables:
#>  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
#>  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
#>  $ disp: num  160 160 108 258 360 ...
#>  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
#>  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
#>  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
#>  $ qsec: num  16.5 17 18.6 19.4 17 ...
#>  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
#>  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
#>  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
#>  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...

# names of the variables in the data frame
names(mtcars)
#>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#> [11] "carb"
length(mtcars)
#> [1] 11
```

Chapter 6

Simple functions on vectors

In this section, we illustrate calculations with a single vector (element by element) and the recycling rule.

```
visits1 <-  c(12, 2, 45, 75, 65, 11, 3)
visits2 <- c(23, 4, 5, 78, 12, 0, 200)

total <- visits1 + visits2
total
#> [1]  35   6  50 153  77  11 203

total.p2 <- total^2
total.p2
#> [1]  1225    36  2500 23409  5929   121 41209

apples <- 15
bananas <- 60
my.fruits <- apples + bananas
my.fruits
#> [1] 75

l3 <- c(12, 34, 50)
l2 <- c(10, 3)
tt <- l3 + 5
tt
#> [1] 17 39 55

ltotal <- l3+l2 # recycling!!!
#> Warning in l3 + l2: longer object length is not a multiple of shorter
#> object length
```

```
ages <- c(28, 33, 39, 56, 34, 45, 27, 40)
ages
#> [1] 28 33 39 56 34 45 27 40
max(ages)
#> [1] 56
sum(ages)
#> [1] 302
length(ages)
#> [1] 8
```

```
ages <- c(28, 33,39,56,34,45, 27,40, NA)
mean(ages, na.rm=TRUE)
#> [1] 37.75

# for help on a command, simply type ? in front of it
?mean
```

Chapter 7

Subsetting

7.1 Generalities about subsetting

There are three subsetting operators: `$`, `[]` and `[[]]`.
Some functions also allow to create subsets: we'll see that later.
We can combine subsetting and assignment to change some parts of an object.
Complement to `str()`.

```
va <- c(13.1, -15.2, 0.3, 2.4, 10.5, -3.6, 9.7) # create the vector va
str(va)
#>  num [1:7] 13.1 -15.2 0.3 2.4 10.5 -3.6 9.7
```

One can see from this call, that va is a simple vector with r length(va) elements. Subsetting means chosing among these.

7.2 []

Applies to vectors, matrices, lists and data frames.
Can be used with:

```
- positive or negative values,
- many values in a vector,
- logical, `NA`,
- character vectors when names.
```

7.2.1 On a vector

Here are a few examples of that object used on a vector.

```
va <- c(13.1, -15.2, 0.3, 2.4, 10.5, -3.6, 9.7)
va[1] # element 1
#> [1] 13.1
va[c(3:5)] # elements 3 to 5
#> [1]  0.3  2.4 10.5
va[-1] # all elements minus the element 1
#> [1] -15.2   0.3   2.4  10.5  -3.6   9.7
va[c(TRUE,TRUE,TRUE,FALSE,FALSE,TRUE,FALSE)]
```

```
#> [1]  13.1 -15.2   0.3  -3.6
va[c(TRUE,FALSE)] # notice the recycling at play here
#> [1] 13.1  0.3 10.5  9.7
va[NA]
#> [1] NA NA NA NA NA NA NA
va[] # nothing selected gives the full vector
#> [1]  13.1 -15.2   0.3   2.4  10.5  -3.6   9.7
names(va)<-letters[1:length(va)] # give names to va
# notice that we subset the vector letters (given by R) and that we don't
# specify the length but give a general value
# now we can subset using names
va
#>     a     b     c     d     e     f     g
#>  13.1 -15.2   0.3   2.4  10.5  -3.6   9.7
va[c("a","e","b")]
#>     a     e     b
#>  13.1  10.5 -15.2
```

7.2.2 On a list

```
mylist<- list(numbers=c(1:20),
              ournames=c("Jim","Jules"),
              results= c(T,F,F,T),
              school=factor(c("Primary", "Secondary", "Tertiary"), ordered=TRUE))
str(mylist)
#> List of 4
#>  $ numbers : int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
#>  $ ournames: chr [1:2] "Jim" "Jules"
#>  $ results : logi [1:4] TRUE FALSE FALSE TRUE
#>  $ school  : Ord.factor w/ 3 levels "Primary"<"Secondary"<..: 1 2 3

mylist[1] # first element of the list
#> $numbers
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
class(mylist[1])
#> [1] "list"
mylist[[3]][3]
#> [1] FALSE
```

Notice that the result of this subsetting is a r class(mylist[1])!

7.2.3 On a matrix

```
set.seed(42)
my.mat <- matrix(floor(runif(30)*10), nrow=5)
my.mat
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    9    5    4    9    9    5
#> [2,]    9    7    7    9    1    3
#> [3,]    2    1    9    1    9    9
#> [4,]    8    6    2    4    9    4
#> [5,]    6    7    4    5    0    8
str(my.mat)
```

```
#>  num [1:5, 1:6] 9 9 2 8 6 5 7 1 6 7 ...
length(my.mat)
#> [1] 30
```

The structure shows that there are r length(dim(my.mat)) dimensions. For subsetting, we must give r length(dim(my.mat)) dimensions! Same rules as for the vectors apply.

```
my.mat[2,3] # 2nd row, 3rd row
#> [1] 7
my.mat[-1,]
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    9    7    7    9    1    3
#> [2,]    2    1    9    1    9    9
#> [3,]    8    6    2    4    9    4
#> [4,]    6    7    4    5    0    8
colnames(my.mat) <- letters[1:ncol(my.mat)] # give names to the columns
rownames(my.mat) <- LETTERS[1:nrow(my.mat)] # give names to the rows
my.mat
#>   a b c d e f
#> A 9 5 4 9 9 5
#> B 9 7 7 9 1 3
#> C 2 1 9 1 9 9
#> D 8 6 2 4 9 4
#> E 6 7 4 5 0 8
my.mat["C",c("a","c","e")]
#> a c e
#> 2 9 9
```

7.3 [[]]

This object is used mainly for lists.

```
mylist<- list(numbers=c(1:20),
              ourname=c("Jim","Jules"),
              results= c(T,F,F,T),
              school=factor(c("Primary", "Secondary", "Tertiary"), ordered=TRUE))
str(mylist)
#> List of 4
#>  $ numbers : int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
#>  $ ournames: chr [1:2] "Jim" "Jules"
#>  $ results : logi [1:4] TRUE FALSE FALSE TRUE
#>  $ school  : Ord.factor w/ 3 levels "Primary"<"Secondary"<..: 1 2 3
mylist[[1]]
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

7.4 $

This object is usually for data frames, where it gives the variable.
It allows partial matching (e.g., `mtcars$gear` is the same as `mtcars$gea`)

```
data(mtcars)
names(mtcars)
#>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#> [11] "carb"
mtcars$mpg
#>  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
#> [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
#> [29] 15.8 19.7 15.0 21.4
mtcars[["mpg"]] # just exactly the same, but R users prefer the $
#>  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
#> [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
#> [29] 15.8 19.7 15.0 21.4
mtcars["mpg"] # NOT the same at all! [] preserves the class
#>                      mpg
#> Mazda RX4           21.0
#> Mazda RX4 Wag       21.0
#> Datsun 710          22.8
#> Hornet 4 Drive      21.4
#> Hornet Sportabout   18.7
#> Valiant             18.1
#> Duster 360          14.3
#> Merc 240D           24.4
#> Merc 230            22.8
#> Merc 280            19.2
#> Merc 280C           17.8
#> Merc 450SE          16.4
#> Merc 450SL          17.3
#> Merc 450SLC         15.2
#> Cadillac Fleetwood  10.4
#> Lincoln Continental 10.4
#> Chrysler Imperial   14.7
#> Fiat 128            32.4
#> Honda Civic         30.4
#> Toyota Corolla      33.9
#> Toyota Corona       21.5
#> Dodge Challenger    15.5
#> AMC Javelin         15.2
#> Camaro Z28          13.3
#> Pontiac Firebird    19.2
#> Fiat X1-9           27.3
#> Porsche 914-2       26.0
#> Lotus Europa        30.4
#> Ford Pantera L      15.8
#> Ferrari Dino        19.7
#> Maserati Bora       15.0
#> Volvo 142E          21.4
class(mtcars["mpg"])
#> [1] "data.frame"
```

7.5 Combining subsetting

Notice that we can often subset further until having the desired subset. Here are a few examples.

```
mtcars$mpg[1:4]
#> [1] 21.0 21.0 22.8 21.4
mylist[["ournames"]][1]
```

```
#> [1] "Jim"
mylist$ournames[1]
#> [1] "Jim"
```

7.6 Subsetting with one condition

We can use conditions for subsetting

```
mtcars[mtcars$cyl==8,]
#>                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
#> Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
#> Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
#> Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
#> Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
#> Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
#> Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
#> Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
#> Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
#> AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
#> Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
#> Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
#> Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
#> Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
mtcars[mtcars$cyl==8 & mtcars$carb==4,]
#>                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
#> Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
#> Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
#> Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
#> Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
#> Ford Pantera L      15.8   8  351 264 4.22 3.170 14.50  0  1    5    4
mtcars[mtcars$cyl==8, c("cyl", "mpg", "wt")]
#>                     cyl  mpg    wt
#> Hornet Sportabout     8 18.7 3.440
#> Duster 360            8 14.3 3.570
#> Merc 450SE            8 16.4 4.070
#> Merc 450SL            8 17.3 3.730
#> Merc 450SLC           8 15.2 3.780
#> Cadillac Fleetwood    8 10.4 5.250
#> Lincoln Continental   8 10.4 5.424
#> Chrysler Imperial     8 14.7 5.345
#> Dodge Challenger      8 15.5 3.520
#> AMC Javelin           8 15.2 3.435
#> Camaro Z28            8 13.3 3.840
#> Pontiac Firebird      8 19.2 3.845
#> Ford Pantera L        8 15.8 3.170
#> Maserati Bora         8 15.0 3.570
```

7.7 Subsetting and assignment

Subsetting can be used to change a part of an object through assignment. Assign `NULL` to delete subset

```
my.mat
#>   a b c d e f
#> A 9 5 4 9 9 5
#> B 9 7 7 9 1 3
#> C 2 1 9 1 9 9
#> D 8 6 2 4 9 4
#> E 6 7 4 5 0 8
my.mat[,1]<-1:nrow(my.mat)
my.mat
#>   a b c d e f
#> A 1 5 4 9 9 5
#> B 2 7 7 9 1 3
#> C 3 1 9 1 9 9
#> D 4 6 2 4 9 4
#> E 5 7 4 5 0 8
mtcars$mpg[1]<-1234
names(mtcars)
#>  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
#> [11] "carb"
mtcars$drat<-NULL # delete variable drat in data frame mtcars
# does not delete in matrix
names(mtcars)
#>  [1] "mpg"  "cyl"  "disp" "hp"   "wt"   "qsec" "vs"   "am"   "gear" "carb"
```

7.8 Using `which()`

`which()` gives the integers that correspond to the boolean (logical) `TRUE`.
This can help subsetting

```
vb<-1500:1530
vb
#>  [1] 1500 1501 1502 1503 1504 1505 1506 1507 1508 1509 1510 1511 1512 1513
#> [15] 1514 1515 1516 1517 1518 1519 1520 1521 1522 1523 1524 1525 1526 1527
#> [29] 1528 1529 1530
which(vb%%5==0) # which is divisible by 5 (modulo is 0) ?
#> [1]  1  6 11 16 21 26 31
# notice that this is asking each element of vb,
# which() reports the positions for which the answer is TRUE
vb[which(vb%%5==0)]
#> [1] 1500 1505 1510 1515 1520 1525 1530
```

7.9 More advanced stuff

7.9.1 Difference between simplifying and preserving

We say ‘preserve’ to say the same structure is maintained when subsetting (e.g., a subset of a data frame remains a data frame).

Simplifying does not keep the structure but gives the simplest output possible.
`drop` argument allows to preserve (`drop=FALSE`) or not (`drop= TRUE`).
`[]` usually preserves, `[[]]` usually simplifies.
To better understand, check the classes:

```
all.equal(class(mylist[1]), class(mylist[[1]]))
#> [1] "1 string mismatch"
class(mylist[1])
#> [1] "list"
class(mylist[[1]])
#> [1] "integer"
all.equal(class(mylist), class(mylist[1]))
#> [1] TRUE
all.equal(class(mylist), class(mylist[[1]]))
#> [1] "1 string mismatch"
```

We see that `[]` has preserved the class of `mylist` (r class(mylist)), while `[[]]` has not! Another striking example is the following.

```
ma2 <- my.mat[1:2,1:3]
class(ma2)
#> [1] "matrix"
ma3 <- my.mat[1,1:3]
class(ma3)
#> [1] "numeric"
```

`ma3` is NOT a matrix anymore!!! This is because one of its dimensions is 1. Losing track of the class when subsetting can generate lots of problems in the middle of a large code. And it is a common source of error! To be sure of keeping the class, we can use `drop=FALSE` (the class will not be dropped to, usually, a vector).

```
ma4 <- my.mat[1,1:3, drop=FALSE]
ma4
#>   a b c
#> A 1 5 4
class(ma4)
#> [1] "matrix"
```

Chapter 8

Conditions

The general purpose of conditions is to control the flow of our code when executed by R.
In R, it builds on statements such as `if` and `else`.

8.1 `if` statement

The simplest form for a condition uses a `if` statement.
The form is then:

```
if (condition) {
# code to be executed if the condition is met
}
```

The key point is that R will run the code until it finds a condition that is met. If it doesn't find any, it continues to the next lines of code.
Here is an example.

```
gains <- c(10, 3,-5,0,-4,12,4)
sum(gains)
#> [1] 20
if (sum(gains) > 0) {
  print("Congratulations, you are winning!")
}
#> [1] "Congratulations, you are winning!"

gains[1] <- -10 # change the first element of gains
if (sum(gains) > 0) {
  print("Congratulations, you are winning!")
} # notice that there is no output, because the condition was not met
```

8.2 `else` statement

What happens when the condition is not met? It's on the user to decide. It can be nothing or... something else!

```
if (condition) {
# code to be executed if the condition is met
} else {
```

```
# code to be executed if the condition is NOT met
}
```

Here is an example.

```
gains<- c(10, 3,-5,0,-4,12,4)
gains[1] <- -10 # change the first element of gains
sum(gains)
#> [1] 0
if (sum(gains) > 0) {
  print("Congratulations, you are winning!")
} else {
  print("You are not winning!")
} # notice that now there is an output!
#> [1] "You are not winning!"
```

8.3 else if statement

`else if` allows us to introduce another condition to our flow of code

```
if (condition_1) {
# code to be executed if the condition_1 is met
} else if (condition_2) {
# code to be executed if the condition_2 is met
} else {
# code to be executed if NEITHER condition_1 NOR condition_2 is met
}
```

We can use as many `else if` statements as we want.
Here is an example with only one `else if`

```
gains<- c(10, 3,-5,0,-4,12,4)
gains[1] <- -10 # change the first element of gains
sum(gains)
#> [1] 0
if (sum(gains) > 0) {
  print("Congratulations, you are winning!")
} else if (sum(gains)==0) {
  print("You just break even!")
} else {
  print("You are losing!")
}
#> [1] "You just break even!"
```

Notice the potential problem of not putting a `else` statement at the end of the conditions system.

8.4 ifelse statement

For **short** conditions, we can use `ifelse`.
The form is

```
ifelse(condition, value if condition met, value if value not met)
```

Here is an example.

```
gains <- c(10, 3,-5,0,-4,12,4)
sign <- ifelse(sum(gains)>=0,"+","-")
sign
#> [1] "+"
```

8.5 Logical operators: &, | and !

Logical operators are used to combine, mix or negate several conditions: - & means AND - | means OR - ! means NOT

De Morgan's laws may help here.

```
(10%%2==0 & 27%%3==0) # equivalent to (TRUE and TRUE), hence TRUE
#> [1] TRUE
TRUE & FALSE
#> [1] FALSE
!TRUE
#> [1] FALSE
!(TRUE & TRUE)
#> [1] FALSE
!(TRUE & !TRUE)
#> [1] TRUE
((10%%2==0 & 27%%2==0)) # equivalent to (TRUE and FALSE), hence FALSE
#> [1] FALSE
((10%%2==0 | 27%%2==0)) # equivalent to (TRUE or FALSE), hence TRUE
#> [1] TRUE
```

8.6 Writing and interpreting a condition

Notice that what R looks for in a condition is a either a TRUE or a FALSE.
If it encounters a TRUE, it executes the commands, otherwise, it doesn't.
Remember there are many ways to obtain one of these two logicals.
Any of these ways will work as a condition.
Here are examples of less trivial ways of writing a condition

```
gains<- c(10, 3,-5,0,-4,12,4)

if (is.numeric(gains)) {
  print("Seems like it is a numeric vector...")
}
#> [1] "Seems like it is a numeric vector..."
# here, is.numeric(gains) evaluates to TRUE, hence, the code is executed!
# the beginner's way would be to replace the condition by
# if (class(gains)=="numeric")

if (!is.factor(gains)) {
  print("Yeah! We avoided the factor...")
}
#> [1] "Yeah! We avoided the factor..."
# the beginner would write if (class(gains)!="factor")
```

Chapter 9

Functions

Everything that happens in R is a function call. If we want R to do something, we must use a function, may it be as simple as a parentheses `(`. If the function is not built-in or coming in a package, one must write it. A function is itself an object. When applied to another object, the function "makes something" to that object and returns an object.

9.1 Simple function

Here is a simple example of a function that returns the square of the object provided.

```r
pow_two <- function(a){
  a^2
}
pow_two(121)
#> [1] 14641
```

9.2 Structure of a simple function

9.2.1 Name of the function

In our example, the name is `pow_two`.
If we want to call the function for further use, type `pow_two()`

9.2.2 Arguments

The arguments are given in the parentheses right after `function`.
Here, we use an argument `a`, which can be any object, e.g., a vector

```r
pow_two(1:10)
#>  [1]   1   4   9  16  25  36  49  64  81 100
```

When using this function, we can see that one argument only must be passed.
A value for the argument is then given in the parentheses when calling the function, e.g., `pow_two(12)`.
A function can have several arguments

9.2.3 Commands

The commands will be executed, if possible.
One command source of error is when the class of the argument provided is NOT compatible with the commands in the function

```
pow_two("hello") # this can't work
#> Error in a^2: non-numeric argument to binary operator
```

9.2.4 Return of a function

The return of a function is given by the last expression executed inside the function.

```
my_f <- function(a){
  a^2
  a^3 # THIS is the last expression executed
}
my_f(5)
#> [1] 125
```

We can use `return()` in the function in order to chose what it returns.

```
my_f2 <- function(a){
  return(a^2)
  a^3
}
my_f2(5)
#> [1] 25
```

A function can return only one object; this is not really a problem as we can always stack everything in a `list()` and have the function return the list.

```
my_f3 <- function(a){
  list(ptwo=a^2, pthree=a^3, pfour=a^4)
}
my_f3(5:20)
#> $ptwo
#>  [1]  25  36  49  64  81 100 121 144 169 196 225 256 289 324 361 400
#>
#> $pthree
#>  [1]  125  216  343  512  729 1000 1331 1728 2197 2744 3375 4096 4913 5832
#> [15] 6859 8000
#>
#> $pfour
#>  [1]    625   1296   2401   4096   6561  10000  14641  20736  28561  38416
#> [11]  50625  65536  83521 104976 130321 160000
```

If we want to use the return of our function, we must assigned it to an object.

```
x <- my_f3(1:5) # evaluates the function above on the vector 1:5
x$pthree # shows the element pthree of the saved list x
#> [1]   1   8  27  64 125
```

An example where we better use `return` is the following

```
pow_three <- function(a){
  if (!is.numeric(a)) {
    return(print("You must give a numeric vector!"))
  } else {
    return(a^3)
  }

}
pow_three(1:4)
#> [1]  1  8 27 64
```

9.3 Multiple arguments and their identification

As mentioned above, a function can have several arguments.

```
my_f4 <- function(abc, xyz){
  c(abc^2, xyz^3)
}
my_f4(5) # this won't work because xyz is missing
#> Error in my_f4(5): argument "xyz" is missing, with no default
my_f4(5, 10)
#> [1]   25 1000
```

R has at least two ways of identifying arguments, by position and by name.
If no name is specified, R uses the arguments according to their positions in the definition of the function.
In the example above, R understands that `abc=5` and `xyz=10` because of the way they are given in the call `my_f4(5, 10)`.
Identification by names would be

```
my_f4(abc=5, xyz=10) # same as before
#> [1]   25 1000
my_f4(xyz=5, abc=10) # different from before because
#> [1] 100 125
# the names are more important than the position
my_f4(xyz=5, 10) # a mix between name and position
#> [1] 100 125
```

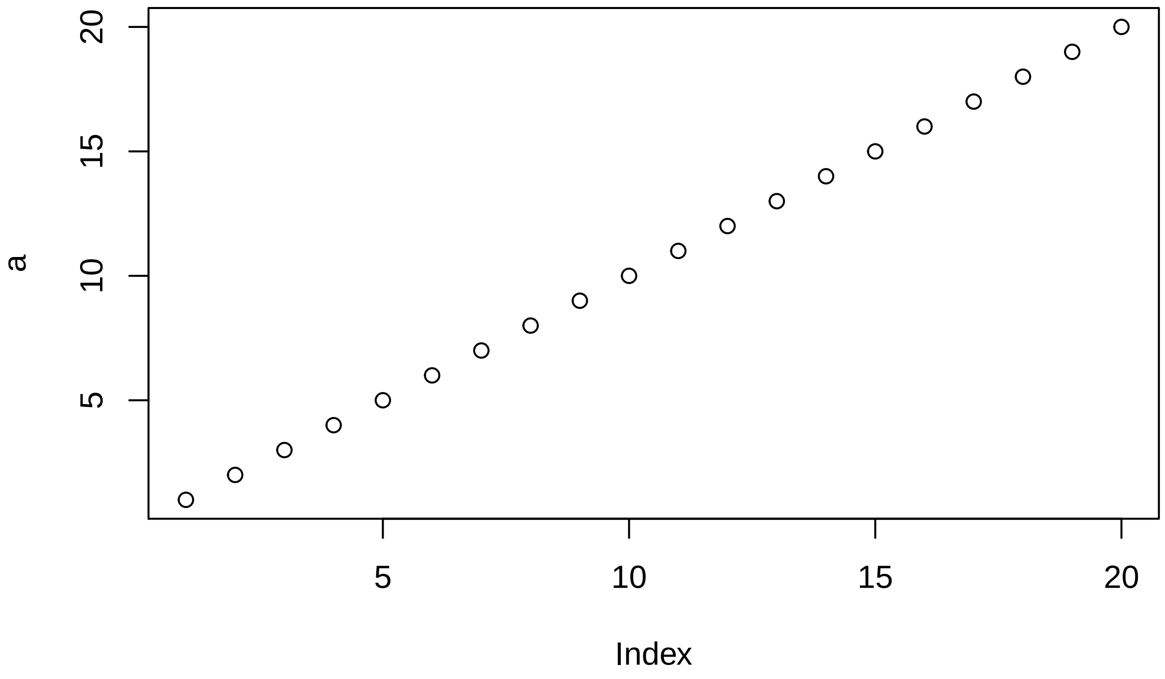
9.3.1 Default values

A function can be defined with default values for their arguments.
R will use the values provided in the call; if one is missing, R will use the default value.

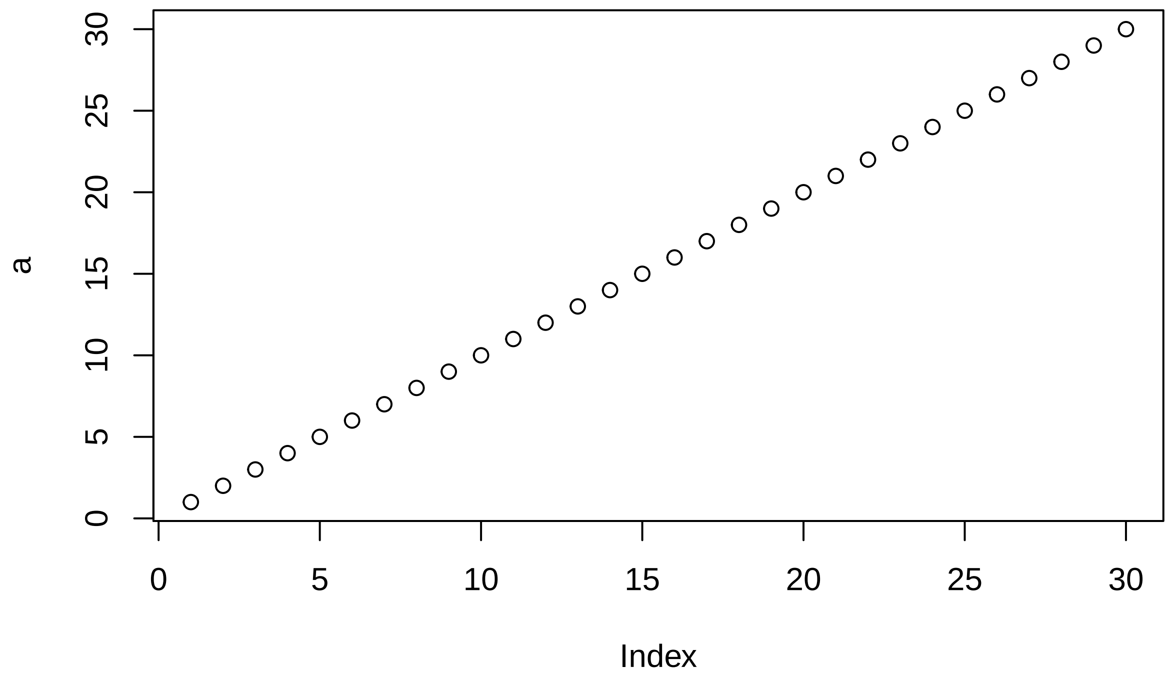
```
my_f4 <- function(abc, xyz=2){
  c(abc^2, xyz^3)
}
my_f4(5) # no second argument? doesn't matter, the function has a
#> [1] 25  8
# default value for it, i.e., 2

my_f5 <- function(a, tada){
  if (tada==TRUE){
    plot(a)
  } else {
    sum(a)
  }
}
```

```
my_f5(1:20, TRUE)
```



```
my_f6 <- function(a, tada=TRUE){
  if (tada==TRUE){
    plot(a)
  } else {
    sum(a)
  }
}
my_f6(1:30)
```



a
0
5
10
15
20
25
30
Index

Chapter 10

Conclusion

We did a great book!

References

Bibliography

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2018). *rmarkdown: Dynamic Documents for R.* R package version 1.11.

Pakin, S. (2009). The comprehensive latex symbol list.

Wickham, H. (2014). *Advanced R.* Chapman and Hall/CRC.

Wickham, H. (2017). *tidyverse: Easily Install and Load the 'Tidyverse'.* R package version 1.2.1.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., and Woo, K. (2018). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics.* R package version 3.1.0.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data.* " O'Reilly Media, Inc.".

Xie, Y. (2018a). *bookdown: Authoring Books and Technical Documents with R Markdown.* R package version 0.9.

Xie, Y. (2018b). *knitr: A General-Purpose Package for Dynamic Report Generation in R.* R package version 1.21.