R beyond data analysis

Adrian Stanciu

2024-06-03

You might know `r` from data analysis. But, `r` can do much more than that for you! This short book supports the seminar with the same title offered by Asst. Prof. Adrian Stanciu at University of Luxembourg.

Table of contents

Preface 6

7

1 General introduction 8
Why this seminar? 8
1.0.1 Reason 1 9
1.0.2 Reason 2 9
1.0.3 Reason 3 10
What else is good to know? 10
1.0.1 Some wizardry stuff 10
1.0.2 GitHub 11
1.0.3 Pushing, pulling, cloning and commiting . 12
1.0.4 GitHub client 13
Illustrative example 14
1.0.1 Background 14
1.0.2 The plan 15

2 R universe 16
R (the console and language) 17
The basics (the very basics!) 17
2.0.1 Objects 18
2.0.2 Vectors 19
2.0.3 Data tables 20
Functions 22
Packages 24
Base R vs. Packages 28
RStudio 28
Advanced resources 30

3 Automatization 31
Elements and structure 32
Knit 34

Live documents 35
3.0.1 Path dependencies 36
3.0.2 The set up 36
3.0.3 Importing data 37
3.0.4 Plain text vs. live text 38
3.0.5 Live text in focus 40
3.0.6 Automated graphs and tables 40
Knit with parameters 45
3.0.1 Example progression 46
3.0.2 Changing setup to parameterized report . 47
3.0.3 Knitting the document 48
3.0.4 Something to do by yourself 49
3.0.5 Another table example 49
Advanced resources 50
3.0.1 Towards shiny apps 50
3.0.2 Quarto 51

4 Self-publish 52
Prepping Quarto 53
Website 53
4.0.1 .qmd 54
4.0.2 .yml 55
4.0.3 .css 57
4.0.4 _site folder 57
4.0.5 Deployment 57
Online book 62
4.0.1 .qmd 62
4.0.2 .yml 62
4.0.3 .bib 64
4.0.4 Deployment 65
Advanced resources 66

5 Shiny apps 67
The set-up 68
R script files 69
Shiny apps structure 71
Code for UI 72
5.0.1 Layout 73
5.0.2 Input types 74
5.0.3 Conditional panels 76
5.0.4 Tabset 77

Code for server 78
5.0.1 Reactive objects 78
5.0.2 Input objects 79
5.0.3 Output objects 80
Run the app locally 81
Deployment 82
(Optional) Push to GitHub 83
Progress illustrative example 83
Advanced resources 87

6 Parting words 88
Advanced resources 89

About the author 90

References 91

Preface

The book is divided into several chapters as follows:

- Chapter Chapter 1: This is an introduction to the purpose and mission of this seminar, and provides useful information to get one started.
- Chapter Chapter 2: This part is a gentle introduction in the `r` universe.
- Chapter Chapter 3: This part addresses the notion of automatization of workflows.
- Chapter Chapter 4: This part provides tips on self-publishing content online.
- Chapter Chapter 5: This part introduces shiny-apps as tools for communicating research output.
- Chapter Chapter 6: This is an overview of the learned material and contains some tips for individual work.

To illustrate the benefits of working with *R* beyond data analysis while transitioning to a holistic work flow, we will build on an example that evolves throughout the chapters Chapter 3–Chapter 5.

1 General introduction

`r` is a programming language that can be used to develop tools that harvest the potential of the Internet while it can help you develop a holistic approach to your work routine.

This seminar aims to only provide a gentle introduction to some of the things that `r` can do for you. After the seminar, those interested can pursue further their interests and use the advanced resources provided to dive deeper into the topics.

Its contents target developing skills that can be applied throughout the university studies as well as on the job market.

Why this seminar?

The are a couple of reasons why someone might be interested in using `r` beyond data analysis.

1.0.1 Reason 1

First, `r` is a programming language thus it is a gateway to other "hard-core" programming languages. For someone who wants to re-invent oneself, `r` can be a useful companion during transitioning from closed-ended data analysis software like SPSS, Mplus or Stata toward a language based logic in data analysis which can be elevated to a holistic work flow.

When using data analyis software like SPSS, Mplus or Stata, one has at one's disposal powerful tools specifically designed to cover the unique purposes of data analysis. This means however also that one needs to simultaneously master a number of other software in writing up, storing and disseminating one's work. A typical work routine involves:

a. importing a dataset in the preferred data analytical software like SPSS,
b. performing the needed analyses,
c. copy/ pasting the result output into a text editor like Word,
d. bouncing forth and back between steps a, b, and c until final results are ready,
e. saving the manuscript as a PDF copy.

Sometime, when one follows the open science practices, there are a couple of extra steps involved:

f. upload the PDF copy to an online repository,
g. upload scripts and material to the online repository (after ensuring a good documentation),

Meanwhile, with the help of `r` one can develop a work routine wherein all the "a" through "g" steps, as well as several other, can be integrated into one work flow in time.

1.0.2 Reason 2

The second reason why one might be interested in `r` beyond data analysis is the appeal of using an open source and community maintained work environment.

This usually means that a dedicated team of specialists develop and maintain software like SPSS, Mplus or Stata which most often than not is available against costs. On the other hand, `r` is open source which means that the code is publically available and everyone can contribute to its development. The Cran website hosts an archive and recent developments.

1.0.3 Reason 3

Third, `r` is a programming language around which a number of excelent tools have been developed. All of these tools, some of which are covered in this seminar, are likewise open source and can be used in an integrated work environment. This means that if one wants to transition to `r`, one can have access to a universe of new posibilities including, for example, creating websites, web applications as well as working seamlessly and simultaneously with several other programming languages including `python` and `SQL`.

What else is good to know?

1.0.1 Some wizardry stuff

Well, if you know programming, it is all too easy. If you don't, nothing makes sense.[1]

One of the first things that one needs to do before using `r` beyond data analysis is to connect all the dots so-to-speak. `r` beyond data analysis relies on an integrated work environment that includes the programming language itself `r`, a work environment interface like `RStudio` as well as online repositories and servers, for general purposes like `GitHub` and for specific purposes like shiny-apps the `shinyapp.io`.

To seamlessly write code and publish it online while ensuring that it does what it is supposed to, there has to be an open channel between all these elements – the work environment needs to be integrated, that is.

[1] I estimate I know about 0,01 %.

To integrate all of these things one needs:

a. an account on these platforms,
b. establishing a communication channel between platforms and working machine (your personal computer),
c. encrypting this communication channel.

I won't cover all the required steps into detail here. This online resource provides everything one needs.

For the sake of simplicity, which happens to be the fundamental piece for the work flow we address in this seminar, one needs to have `Git` installed on one's local machine. You might already have it, so please check it first. Note that installing `Git` might take some time, so don't be surprised if that happens.

– Install for Windows by downloading from `https://gitforwindows.org/` (here).

– Install for Mac or Linux using `Homebrew`. Follow the steps here `https://brew.sh/`(here).

Tip 1: About Git

`Git` is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency https://git-scm.com/.

1.0.2 GitHub

Technically, this section and the one above are difficult to tease apart. For those interested, this online resource can answer further questions and is a good starting point for an advanced workflow with `r`, `Git` and `GitHub`.

`GitHub` is an online platform that facilitates collaboration, storage and publishing of almost anything programming-related. It is an online and publically accessible repository in that anyone with an account can create repositories, upload and download codes and projects. Basically it is `facebook` for programmers.

A relative of `GitHub` is `GitLab` which is specifically designed for internal use in institutions. If you want to publish your code online (website and online books, for example) and make it accessible to everyone in the world then you should use `GitHub`. If however, you'd like to work on a project internally, only with colleagues from your institution (or other registred institutions) you should use `GitLab`, which is available through your institutions. At the University of Luxembourg, there is a designated `GitLab` platform.

Open an account on https://github.com/.[2]

After the `GitHub` account is live, the next step is to open and encrypt the communication channel between your local machine and your `GitHub` repository account. This step can be tricky, so take your time and equip yourself with lots of patience. All the steps can be found here. A simplified, and somewhat visual, description is provided also here[3]

1.0.3 Pushing, pulling, cloning and commiting

It is helpful to understand first the concepts of `pushing`, `pulling`, `cloning` and `committing`. These are verbs in the English language thus they indicate actions that one can do. These actions all are in reference to the code one writes and the current location of the code and where one wants the code to be placed.

- `pushing` is basically the action of uploading the written code or files from the local machine onto the online repository through the distribution control system, `Git` that is. `pushing` only has a resemblance to uploading because pushing a code onto an online repository means simultaneously uploading it and creating a history of the code in the project. In some cases, pushing a code also means it

[2]Note that it is not quite clear where the data is stored on these servers. So, if you are concerned about data protection issues, be sure you do not upload sensitive information. For the sake of this seminar this is not an issue, but be warned!

[3]This resource is also a step-by-step guide for creating a website using `r` and associated tools. This will be covered is Chapter 4 of this short book.

"activates its" functions. In chapter Chapter 4, for example, we will see that the written code on the local machine becomes a book or a website once it is `pushed` onto the `GitHub` online repository.

- `pulling` is in many ways the opposite of the `pushing` verb. In this case, one is downloading the code or files from the online repository on the local machine. This can come in handy when one is working with others on a common project, and, while gone, someone else has updated the project; Someone else has `pushed` a code update, for example. Also, this is a useful thing when one uses variant machines or when one has deleted by mistake the project from the local machine, which can happen!

- `cloning` is in many ways copy-pasting a repository from the online `GitHub` server to the local machine. The outcome is a straightforward one: Cloning a repository to the local machine means also that the history, code changes and dependencies are reproduced on the local machine.

- `committing` is exactly what you think it might mean in the English language – to commit to something or someone has a *finality* aspect to it, or enduring, or fixed, if you will. When writing code or changing code (or files for that matter) on the local machine, you commit it to your project when you are happy with it. This then means that the updated code is now integrated in the project, it can be traced backwards in the history of the project. The nice thing about working this way is that once a code update or file is `committed` to the project, the project itself is updated/ modified accordingly.

1.0.4 GitHub client

Writing code and creating `Rmarkdown` files on the local machine is rather straightforward. For that, one needs only an `r` client, and typically `RStudio` (download here, will be covered in more detail in Chapter 2) is the preferred one.

However, as soon as online repositories, collaborative work and co. become relevant, one needs to communicate with these non-local machines.

One way to do this is through line coding in `git`, which can be accessed via the `Terminal` in the `RStudio`. This can be straightforward and eventually becomes a routine. This cheet sheet provides all the necessary `git`commands.

Meanwhile, if one prefers using `git` through a friendlier visual interface, then one would want a `GitHub` client. GitHub Desktop can be downloaded for free and has a simple interface. Check this resource to getting started with GitHub Desktop.

Illustrative example

1.0.1 Background

Throughout Chapter 3–Chapter 5, we will use a sample of the data reported in Stanciu et al. (2017)[4].

Stanciu et al. (2017) studied how people stereotyped varying social groups in terms of warmth and competence across several regions in Romania. For this seminar, we will use data from a sample of $n = 100$ participants selected at random from the reported data set.

The data includes the following variables:

- `ppn` participant number,
- `gen` self-reported gender of participant as female (1) or male (2),
- `age` chronological age as it was self-reported in years,
- `res` region or residence of the participant,
- `res_other` open ended question regarding region or residence of the participant,
- `men_warm` participant's stereotypeical evaluation of men in terms of warmth,

[4]The article can be downloaded also via Orbilu at the University of Luxembourg. See here

- `men_comp` participant's stereotypeical evaluation of men in terms of competence,
- `wom_warm` participant's stereotypeical evaluation of women in terms of warmth,
- `wom_comp` participant's stereotypeical evaluation of women in terms of competence.

Stereotypical evaluations were assessed on Likert scales with these answer options:

1 = strongly disagree, 2 = disagree, 3 = undecided, 4 = agree, 5 = strongly agree.

Tip 2: Access data

Data and meta-data referred throughout this short book are downloadable directly from inside this book. Navigate to the left panel of the book, and press the download icon under book title.
Likewise, all `r` scripts, `.Rmd` and `.qmd` illustrative examples are provideed in `.zip` compressed files.

1.0.2 The plan

In Chapter 3 we will use this sample to illustrate how certain steps in working with data can be automatized. We will write static text and "living" texts whereby we use `r` code to populate text dynamically with information automatically retrieved directly from data.

In Chapter 4 we will see how the work from previous chapter can be integrated in a self-published book or as content for the personal website. We will focus on creating tables and graphs using the sample.

In Chapter 5 we will see how we can present results in an interactive manner using online applications. We will focus on how to create tables and graphs as well as "live" texts for the online app.

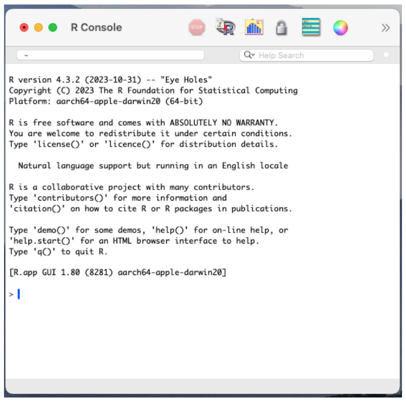
2 R universe

To paraphrase, *R* is a dialect of another programming language, namely *S*. You can read more about the history of *R* (and *S*) here. Long story short, *R* is a programming language derived from *S* that was available only for commercial packages. *R* was created by **Ross Ihaka** and **Robert Gentleman** in 1991 at the University of Auckland, New Zealand. In 1995, it became an open source code thanks to contributions by **Martin Mächler**.

In this short book, I will use interchangeably `r` and *R*.

The online and free book by **Roger D. Peng** *R programming for data science* is a good further reading for those interested.

The online and free book by **Oscar Baruffa** *Big book of R* is an excelent collection of available resources to learn and master *R*.



```
R version 4.3.2 (2023-10-31) -- "Eye Holes"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20 (64-bit)

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

  Natural language support but running in an English locale

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

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

[R.app GUI 1.80 (8281) aarch64-apple-darwin20]

> 
```

Figure 2.1: The *R* console

R (the console and language)

When most people talk about `r` they mean both the programming language and a console. Unless they are IT experts who can make the distinction with ease. But, for the purpose of the seminar, or as a typical `r`-user for what is worth, it really doesn't matter.

When working with `r` one needs a designated console for writing the code, and this is easy to detect as `r`-console (see Figure Figure 2.1).

To download *R*, go to the cran website and select the file suitable for your operating system. Unzip or install that file and the `r`-console will be installed on your machine.

The basics (the very basics!)

This seminar will cover only the very absolute basics of working in `r`. Designated courses are available at the university and elsewhere as part of summarschools or workshops. Of course, one can learn `r` using the freely available online content. Use YouTube and Google for that. For example, this online resource is a good starting point.

The first thing to notice in the `r` console is the symbol `>` followed by the text placer `|`. This specifies the line where to write the `r` code.

Once the code is written and the key Enter is pressed, the code basically `run`s or is computed by the machine which returns an outcome. (Here the symbol `>` is not visible but the outcome line can be identified through the use of squared brackets[...])

```
2+2
```

```
[1] 4
```

2.0.1 Objects

It is useful to work with objects in `r`. That is, whatever code you write, place it into an object and then run the object itself. See below.

```
# no object created
2+2
```

```
[1] 4
```

```
# object is first created and then run
sum<-2+2
sum
```

```
[1] 4
```

Using objects simplifies a lot the work flow because you can combine objects in any way you can imagine!

```
# creates a second object called mean
mean<-mean(c(1,2,5,7,8,9))
mean
```

```
[1] 5.333333
```

```
# and then adds the two objects 'sum' and 'mean' together
result<-sum+mean
result
```

```
[1] 9.333333
```

2.0.2 Vectors

There are multiple types of objects that one can create in `r`. The most important ones are vectors and data tables.

For simplicity reasons, vectors can be numeric, character strings or logical. A vector is scalable meaning that it can hold up to a gazilion of elements.

```
# example of numeric vectors
vec1<-c(1,3,66,9,121)
vec1
```

```
[1]   1   3  66   9 121
```

```
# example of character string vector
vec2<-c("A","Ab","This or that","C","d")
vec2
```

```
[1] "A"            "Ab"           "This or that" "C"            "d"
```

```
# example of logical vector
vec3<-c(TRUE,TRUE, FALSE, TRUE)
vec3
```

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

One can do all sorts of things with and to vectors. See for example here.

2.0.3 Data tables

Data tables combine multiple vectors. Data tables can combine all sorts of vectors and can have varying internal structures. When one downloads (or uses one own's) dataset, that is typically a data table in a specific format, `.sav` for SPSS or `.xlsx` for Microsoft Excell. Data formats can also be `.dat`, `.csv`, `.asci` and so on.

A data table in `r` comprises multiple vectors and involves an organization wherein typically rows represent entries in the data table and columns represent vectors of the data table. In other words, rows represent cases and columns represent variables.

```
# create a simple data table
df<-data.frame(col1=vec1,
                  col2=vec2)
df
```

```
  col1         col2
1    1            A
2    3           Ab
3   66 This or that
4    9            C
5  121            d
```

```
# one can then access the varying elements of the data table

# access col1
df[,1]
```

```
[1]   1   3  66   9 121
```

```
# access first row
df[1,]
```

```
  col1 col2
1    1    A
```

```r
# access entry at first row and col1
df[1,1]
```

```
[1] 1
```

One can perform all sorts of actions on the data table as a whole or on elements of the data table.

```r
# checks the elements of the data table
str(df)
```

```
'data.frame':	5 obs. of  2 variables:
 $ col1: num  1 3 66 9 121
 $ col2: chr  "A" "Ab" "This or that" "C" ...
```

One can see that col1 is a numeric `num` vector and col2 is a character string `char` vector.

```r
# provides a summary of the data table
summary(df)
```

```
      col1           col2          
 Min.   :  1   Length:5          
 1st Qu.:  3   Class :character  
 Median :  9   Mode  :character  
 Mean   : 40                     
 3rd Qu.: 66                     
 Max.   :121                     
```

One can see that different summary stats are available for `num` and `chr` vectors.

```r
# performs an addition on the numeric vector of the data table
df[,1]+100
```

```
[1] 101 103 166 109 221
```

Functions

To be entirely honest, `r` functions are something a bit advanced. But, some rudimentary functions can be written by beginners too. The trick is to figure out what is repetitive in the code that one wants to write. This logic proves useful when one needs to apply a command on a number of objects for an undetermined number of times.

Functions are easy to spot in *R* because they are labeled as such and have a unique code structure: `function(){}`.

The rule of thumb is `()` defines the elements that are fed into the function while `{}` contains the function itself.

Here is an example. We use a dataset that comes pre-installed with *R* (`iris`), perform an addition on all the numerical variables and then write a function to simplify the task.

```r
# see the first ten rows of the pre-installed dataset iris
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```r
# numerical columns are then columns 1 through 4

# adds 3 to all numerical columns
head(iris[,1:4] + 3)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          8.1         6.5          4.4         3.2
2          7.9         6.0          4.4         3.2
3          7.7         6.2          4.3         3.2
4          7.6         6.1          4.5         3.2
5          8.0         6.6          4.4         3.2
6          8.4         6.9          4.7         3.4
```

```
# add 77 to all numerical columns
head(iris[,1:4] + 77)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1         82.1        80.5         78.4        77.2
2         81.9        80.0         78.4        77.2
3         81.7        80.2         78.3        77.2
4         81.6        80.1         78.5        77.2
5         82.0        80.6         78.4        77.2
6         82.4        80.9         78.7        77.4
```

```
# write a function
# this function takes two arguments: a dataset 'df' and a constant 'n'
func1<-function(df,n){

  tmp <- Filter(is.numeric, df) # we first filter the dataframe for numeric columns

  tmp + n # we then add the constant to all the numeric columns
}


# we apply the function and add 3 to all numeric columns of iris
# we only ask to see the first ten rows of the outcome using head()
head(func1(iris,3))
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          8.1         6.5          4.4         3.2
2          7.9         6.0          4.4         3.2
3          7.7         6.2          4.3         3.2
4          7.6         6.1          4.5         3.2
5          8.0         6.6          4.4         3.2
6          8.4         6.9          4.7         3.4
```

```
# we apply the function and add 99 to all numeric columns of another pre-installed dataset 'mt
# we only ask to see the first ten rows of the outcome using head()
head(func1(mtcars,99))
```

```
                    mpg cyl disp  hp   drat     wt   qsec  vs  am gear carb
```

```
Mazda RX4         120.0 105  259 209 102.90 101.620 115.46  99 100  103  103
Mazda RX4 Wag     120.0 105  259 209 102.90 101.875 116.02  99 100  103  103
Datsun 710        121.8 103  207 192 102.85 101.320 117.61 100 100  103  100
Hornet 4 Drive    120.4 105  357 209 102.08 102.215 118.44 100  99  102  100
Hornet Sportabout 117.7 107  459 274 102.15 102.440 116.02  99  99  102  101
Valiant           117.1 105  324 204 101.76 102.460 119.22 100  99  102  100
```

Packages

An *R* package contains code, documentation, and sometimes even data. These packages are developed to serve a specific purpose such as simplifying a work routine or perform advanced computational routines. Packages can be downloaded for free and then immediately used. Of course, everyone can write an *R* package, which of course is not a easy thing to do. But if at any point and for whatever reason you need to, then know that it is possible.

Everything one needs to know about packages can be found in this comprehensive book by Hadley Wickham[1] and Jennifer Bryan.

`r` packages use the philosophy of working with functions to simplify otherwise highly complex code. Some of the fundamental packages to start with are `tidyverse` (for data preparation and manipulation but also contains several other useful packages like `ggplot2` for creating graphics). Other packages that are the focus of this seminar are `rmarkdown` (the fundamentals of Chapter 3 through Chapter 5), `quarto` (needed for self-publishing books and website; covered in Chapter 4),`tinytex` (for latex distributions aka. creating PDFs), `shiny` (for web applications; covered in Chapter 5).

What you absolutely need to know about packages is that the vast majority do not come pre-installed with the `r` console but can be installed by request. Installing any package in *R* follows this basic routine:

[1]He is THE `r` expert. See his website.

```r
# installs `tidyverse`
 install.packages("tidyverse")

# makes it available for R on your local machine
# this step is crucial if you want to have access to all the containing function
library(tidyverse)
```

One trick that I think it is absolutely simple to use but can save you a lot of nerves is using the package `pacman` to install any other packages. The nice thing about it is that `pacman` can first check if a package is already installed on the local machine and if not, it downloads it and installs it from `Cran`.

We can now install the basic packages needed for the seminar and mentioned above.

```r
# first, we install the `pacman` package
install.packages("pacman")

# then, we use the function `p_load` from the `pacman` package to install `tidyverse`, `rmarkd
pacman::p_load(tidyverse,rmarkdown,bookdown,quarto,shiny)
```

Tip 3: R Packages with websites

(Almost) Every package has a designated website. Visit the package website for examples on how to use and also to identify the functions contained. For example https://www.tidyverse.org/

Tip 4: R Packages documentation

Call the package documentation by typing in a question mark followed by the name of the package or function contained in a package. For example ?tidyverse

Let's see as an example how the function `filter` from the universe of packages `tidyverse` works. Before that, I want to introduce the pipe operator `%>%`[2] which is instrumental for `r` users. And it simplifies a lot the work flow!

[2]The pipe operator itself is introduced most comonly in the package `dplyr`

%>% follows the logic of, simply and un-elegantly put, “work that happens in the background until the desired output is retrieved”. It also means that using %>% you can compress into one code otherwise a long chain of steps that involve creating objects which are then subjected to new operations.

```
# apply the function filter to the dataset mtcars
# we filter the column cyl such that only cars with a cyl < 5 are displayed
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
```

```
mtcars %>% filter(cyl < 5)
```

```
                mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
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
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
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
Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

```
# we filter the column cyl such that only cars with a cyl exactly equal to 8 are displayed
mtcars %>% filter(cyl == 8)
```

contained in the universe of packages `tidyverse`. But, it can be used differently in other packages too.

```
                     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
```

As an example of the usefulness of the pipeline operator `%>%`, let us apply a double filter. First, on the column `cyl` and then on the column horse power `hp`.

```r
# without the pipeline operator
a<-mtcars %>% filter(cyl < 5)
b<-a %>% filter(hp > 100)
b
```

```
              mpg cyl  disp  hp drat    wt qsec vs am gear carb
Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
Volvo 142E   21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

```r
# with the pipeline operator
mtcars %>% filter(cyl < 5) %>% filter(hp > 100)
```

```
              mpg cyl  disp  hp drat    wt qsec vs am gear carb
Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
Volvo 142E   21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

Of course, this example is too simplistic but imagine having to write a gazillion of lines of code when you could reduce that to a couple. Throughout the seminar we use the pipe operator `%>%` almost everywhere!

Base R vs. Packages

Figure 2.2

A fair warning!

Base *R* is complex but stable. Packages are simple to use but depend on the community for their maintenance. So, the decision is to use something complex but stable or simple but unstable.

For the purpose of this seminar, and for most of the things a regular *R*-user needs, working with packages is indeed the way to go.

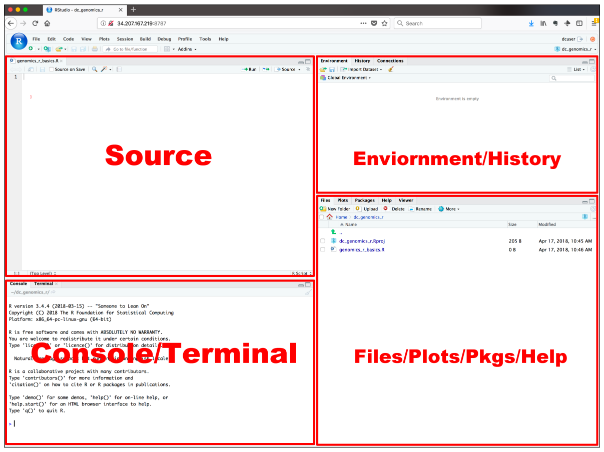
If at any point, you are concerned that the package(s) you use can get outdated, I recommend using the [sic!] package `groundhog` which ensures reproducible code. This package basically goes back in time and installs on the local machine the desired version of the package.

See how it works on this website.

RStudio

In Figure 2.3 you can see the four panels of `RStudio`, the (a) Console/terminal, (b) Source, (c) Environment/history, and (d) Files/plot/packages/help.

- **a) Console/terminal** Here is where the `r` console is integrated in `RStudio`. You can type in your code, have your results previewed, as well as any errors (those happen quite a lot) that occur in your coding.



```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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

  Natural language support but running in an English locale

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

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

> 
```

Figure 2.3: RStudio panels

- **b) Source** This panel is where we will do most of the work throughout the seminar. Think of this panel as the notebook – you write, you draw, you comment on your own work, etc. This panel allows you to communicate with the source material, which can be `r` (the language), `html` (the language) and also lets you populate with content the files needed for the website, for instance.
- **c) Environment/history** This panel is a place where you can see the history of your work. It saves for you the code you ran (either in the console or source panels) and also it contains sort of short-cuts to any data-related work you might have done.
- **d) Files/plot/packages/help** This panel allows you to preview what you've communicated to the machine (laptop) to do. You will note there are several tabs, but the most important one for the seminar are:
  - `Files` is sort of *Windows explorer* in Windows or *Finder* on Mac OS. It is here that you can navigate between folders on the local machine, delete, rename, or more. Here you can also open files in the source panel.
  - `Packages` gives you an overview of packages that are installed and active on the local machine.

    – `Help` is, well, where you will see helpful information about a function or package.

On this youtube channel there is a helpful beginners guide on *R* and `RStudio`. Take some time to familiarize yourself with them.

> Tip 5: Learning resources
>
> If your `RStudio` version is 2024.04. or newer, you should note in the Environment/ History panel a tab "Turorial". That panel contains tutorials for working in *R*. Install first the package `learnr` as indicated and let yourself guided through a number of interactive exercises.

Advanced resources

Together with a colleague, Dr. Ranjit SINGH from GESIS - Leibniz Institute for the Social Sciences, I prepared a workshop on `r` for beginners. All the material is open access via `GitHub`.

You can `clone` the repository on your local machine and do all the exercises.

Navigate first to the page of the repository and then `clone` it to your local machine: https://github.com/adrianvstanciu/rworkshop_open.

3 Automatization

Automatization of the work flow, as I mean it throughout the seminar, is a holistic or integrated approach to the work flow whereby repetitive tasks can be reduced through the use of code and a designated work environment.

This approach can be useful to reduce working time, ensuring reproducible outcome as well as enhancing work transparency.

Among the domains where this is an asset are:

a) Research: Data analysis and results interpretation, writing manuscripts, and adhering to open science.
b) Applied sector: Writing of repetitive reports.
c) Education: Transparent homework.

This chapter covers the basics of creating and working in an integrated work environment.

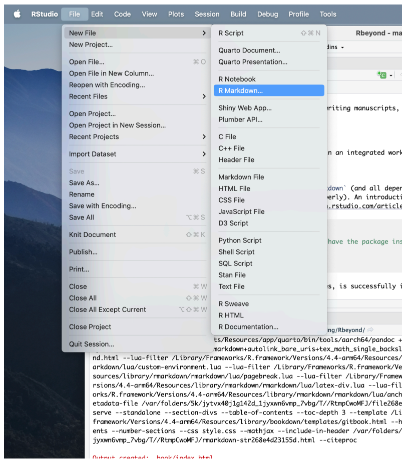
Elements and structure

The first thing we need is to install the package `rmarkdown` (and all dependencies, meaning all the packages that `rmarkdown` needs to function properly). An introduction to R Markdown is available on the official website[1].

```r
# installs package `rmarkdown` and all dependencies
install.packages("rmarkdown", dependencies = TRUE)
```

After the package `rmarkdown`, including its dependencies, is successfully installed, we should be able to create and work in .Rmd files.

Create your first `.Rmd` file and give it a name. For example, `example.Rmd`. You should see it now lower-right panel Files/Packages/Help.



```
ts/Resources/app/quarto/bin/tools/aarch64/pandoc +
markdown+autolink_bare_uris+tex_math_single_backsl
nd.html --lua-filter /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/
arkdown/lua/custom-environment.lua --lua-filter /Library/Frameworks/R.framework/Ve
sources/library/rmarkdown/rmarkdown/lua/pagebreak.lua --lua-filter /Library/Framew
rsions/4.4-arm64/Resources/library/rmarkdown/rmarkdown/lua/latex-div.lua --lua-fil
orks/R.framework/Versions/4.4-arm64/Resources/library/rmarkdown/rmarkdown/lua/anch
etadata-file /var/folders/5k/jytvx40j1g142d_1jyxwn6vmp_7vbg/T//RtmpCwoMFJ/file268e
serve --standalone --section-divs --table-of-contents --toc-depth 3 --template /Li
framework/Versions/4.4-arm64/Resources/library/bookdown/templates/gitbook.html --h
ents --number-sections --css style.css --mathjax --include-in-header /var/folders/
jyxwn6vmp_7vbg/T//RtmpCwoMFJ/rmarkdown-str268e4d23155d.html --citeproc

Output created: _book/index.html
```

Figure 3.1: Create your first RMarkdown file

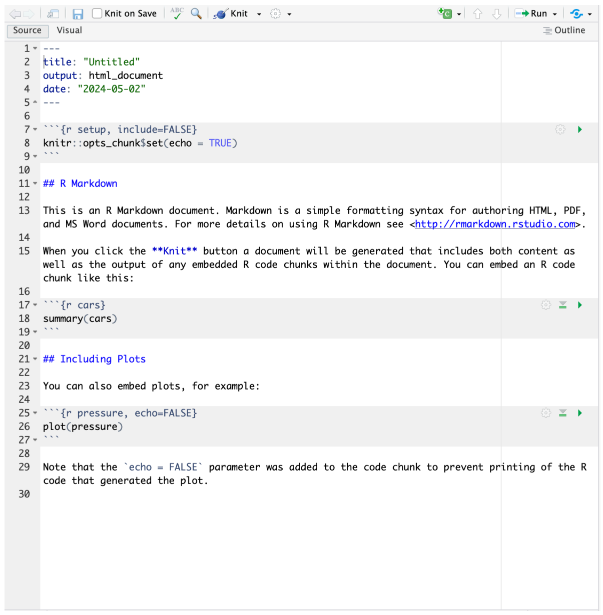
Tip 6: Workflow simplified with projects

I recommend always working with `R projects` which makes it easy for code dependencies and gives structure to

[1]RMarkdown is by now old generation. The new generation is `quarto`, covered in Chapter 4, which keeps all the RMarkdown traits while it simplifies even more the user-experience.

your work flow. To associate your project to an `.Rproj`, go to File and create a New project. Make sure you associate the current folder you are working in with the `Rproj`. This guide can help further.

Once the `.Rmd` has been created, you should be able to open and edit it. It should look like this.



```
 1 ---
 2 title: "Untitled"
 3 output: html_document
 4 date: "2024-05-02"
 5 ---
 6
 7 ```{r setup, include=FALSE}
 8 knitr::opts_chunk$set(echo = TRUE)
 9 ```
10
11 ## R Markdown
12
13 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF,
   and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.
14
15 When you click the **Knit** button a document will be generated that includes both content as
   well as the output of any embedded R code chunks within the document. You can embed an R code
   chunk like this:
16
17 ```{r cars}
18 summary(cars)
19 ```
20
21 ## Including Plots
22
23 You can also embed plots, for example:
24
25 ```{r pressure, echo=FALSE}
26 plot(pressure)
27 ```
28
29 Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R
   code that generated the plot.
30
```

Figure 3.2: RMarkdown document

In Figure 3.2 there are numbered lines from 1 to 30. These indicate lines in the `rmarkdown` file. We refer to these to discuss elements of the file.

- **Lines 1 through 5** (note the `---`) hold the `yaml` header element. Here is where you indicate the characteristics of the entire `rmarkdown` document, and it includes for example the kind of `output` file you want generated, `title` of the document or `date` of the document version. Many other `yaml` attributes exist and depending on your goals, you can easily find them online using a version of the search string `yaml rmarkdown attributes`.
- **Lines 7 through 9** (as well as 17-20 and 25-27) are

`code chunks`. In this case, they are `r` code chunks, as signaled by the small letter `r`. Code chunks are what makes `rmarkdown` documents so powerful. Integrating code chunks into text facilitates the creation of live documents. In other words, plain text merged with code. Each code chunk has multiple attributes that can modify the way the output of the code is integrated and presented in the output document. Note for example at line 7 `{r setup, include=FALSE}`.

- `setup` is the name of the code chunk and this is extremely helpful to give structure to the document but also when cross-referencing figures and tables in the document.
- `,` the comma is crucial here because it signals that what follows are settings of the code ouput
- `include=FALSE` is one such setting attribute and indicates that the code output is not integrated in the output document BUT it runs in the background.

Knit

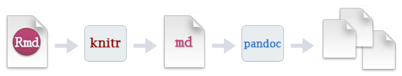


Figure 3.3

Figure 3.3 is an overview of the workflow from an `.Rmd` (editable `rmarkdown` document) to an output document which can be `.pdf`, `.html`, `.docx` and so on. One crucial step happens during the **knit** of `.Rmd` file. Basically, in this step you *knit* everything from the `.Rmd` document together. The name comes from knitting, which is, well, creating something nice and creative from nothing.

In fact, the very Figure 3.4 above was knited to the final document.

The `knit` function comes pre-installed with `RStudio` and can be found in the source panel. Identify it, and knit your first `.Rmd` file first into a `.pdf` and then into a `.html` file.

Figure 3.4: Knited bike cover

Tip 7: Latex distributions for PDF

If knit to PDF didn't work, it might be because we need a latex distribution on the machine that `r` can work with. Try installing the package `tinytex` (Read more on https://yihui.org/tinytex/)

```
# to install tinytex distribution
install.packages('tinytex')
tinytex::install_tinytex()
# to uninstall TinyTeX, run tinytex::uninstall_tinytex()
```

Once you `knit` the `.Rmd` file, a new file will be created in the designated folder. Voilà - you just created your first PDF and/ or HTML document.

Live documents

For the purpose of this seminar, I call live documents those documents that are coded to retrieve data and/ or information from external source material (e.g., datasets or meta-data such as from Excel sheets). This is the building block for creating all sorts of automatized reports.

3.0.1 Path dependencies

Since we are working inside an `.Rproj`, all dependencies are already set up. This helps because when interacting with external source material we need to specify where the code should look for it. This is possible also without working in an `.Rproj` but the dependencies are increasingly more complex to set up. For one, the migration from one operating system to the other may break these dependecies. Furthermore, you need to find the file path on the local machine and then include it in the code. Note that in case of deploying (uploading) the project on `GitHub` the code will break because, of course, the dependencies are only locally relevant. If, however, we work in `.Rproj` and we push all the project files on `GitHub`, then we need not worry about file paths, these are by default set up through the use of the project.

Return to the first `.Rmd` we previously created. Open it in `RStudio`.

Once opened, we may choose to delete the default content, leaving only the `yaml` header intact. Or we may choose to keep the default content. I choose to delete it for the sake of simplicity.

3.0.2 The set up

We can start by setting up the work environment. This means, we should first install the packages we'd need in the process.

We install using `pacman` the packages `tidyverse`, `readxl` (for reading Excel sheets), `haven` (for reading SPSS files), `sjlabelled` (for dealing with labelled dataframes), `kable` and `kableExtra` (for creating tables).

```
install.packages("pacman")

pacman::p_load(tidyverse,readxl,haven,sjlabelled,kable, kableExtra)
# note: this exact code chunk might end up looking differently in the short book
# this is becauase i'd install packages as needed
```

3.0.3 Importing data

Next, we import our dataset in `.sav` format and the Excel sheet in `.xlsx` format. This will allow us access to the contents of those external source material which we can integrate in our final document.

To import these external source material, remember to use objects to store that information. In other words, we import the source material and assing it to objects that we can then perform varying actions onto.

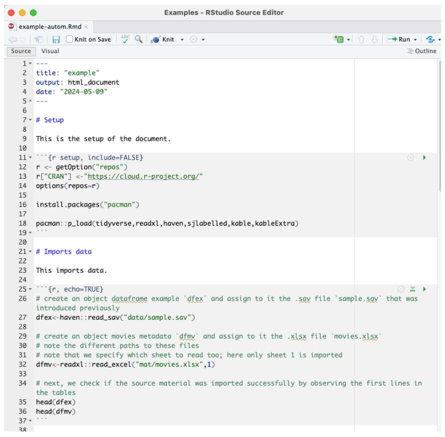
```r
# create an object dataframe example `dfex` and assign to it the .sav file `sample.sav` that wa
dfex<-haven::read_sav("data/sample.sav")

# create an object movies metadata `dfmv` and assign to it the .xlsx file `movies.xlsx`
# note the different paths to these files
# note that we specify which sheet to read too; here only sheet 1 is imported
dfmv<-readxl::read_excel("mat/movies.xlsx",1)

# next, we check if the source material was imported successfully by observing the first lines
head(dfex)
```

```
# A tibble: 6 x 9
    ppn gen           age res       res_other men_warm men_comp wom_warm wom_comp
  <dbl> <dbl+lbl>   <dbl> <dbl+lbl> <chr>     <dbl+lb> <dbl+lb> <dbl+lb> <dbl+lb>
1   459 1 [Female]    24   5 [Iasi] -99         3 [Und~  4 [Agr~ 3 [Unde~ 4 [Agre~
2   592 2 [Male]      21   5 [Iasi] -99         3 [Und~  4 [Agr~ 3 [Unde~ 3 [Unde~
3   634 2 [Male]      21 NA         petrosani   4 [Agr~  5 [Str~ 4 [Agre~ 4 [Agre~
4   369 1 [Female]    30   8 [Gala~ -99       NA        NA       4 [Agre~ 4 [Agre~
5   121 1 [Female]    21   4 [Timi~ -99         4 [Agr~  3 [Und~ 3 [Unde~ 4 [Agre~
6   127 1 [Female]    20   4 [Timi~ -99         4 [Agr~  4 [Agr~ 4 [Agre~ 2 [Disa~
```

```r
head(dfmv)
```

```
# A tibble: 4 x 6
  Movie                        Actor                  Like  Why      Grade Wikilink
  <chr>                        <chr>                  <chr> <chr>    <dbl> <chr>
1 John Wick                    Keanu Reeves           Yes   Fight ~     10 https:/~
2 Call me by your name         Timothee Chalamet      Yes   Beauti~     10 https:/~
```

```
3 Terminator                     Arnold Schwarzenegger Yes   Arnold      9 https:/~
4 4 months 3 weeks and 2 days <NA>                  Yes   Portra~     8 https:/~
```



~~~
---
title: "example"
output: html_document
date: "2024-05-09"
---

# Setup

This is the setup of the document.

```{r setup, include=FALSE}
r <- getOption("repos")
r["CRAN"] <-"https://cloud.r-project.org/"
options(repos=r)

install.packages("pacman")

pacman::p_load(tidyverse,readxl,haven,sjlabelled,kable,kableExtra)
```

# Imports data

This imports data.

```{r, echo=TRUE}
# create an object dataframe example `dfex` and assign to it the .sav file `sample.sav` that was introduced previously
dfex<-haven::read_sav("data/sample.sav")

# create an object movies metadata `dfmv` and assign to it the .xlsx file `movies.xlsx`
# note the different paths to these files
# note that we specify which sheet to read too; here only sheet 1 is imported
dfmv<-readxl::read_excel("mat/movies.xlsx",1)

# next, we check if the source material was imported successfully by observing the first lines in the tables
head(dfex)
head(dfmv)
```
~~~

Figure 3.5: Example .Rmd file

If all went well, your `.Rmd` should look similar to mine (see Figure 3.5).

3.0.4 Plain text vs. live text

In some ways, what we have coded thus far is also an automatized work routine in that the `.Rmd` document automatically retrieves the external source material every single time when it is `knit`-ed into a PDF or HTML file.

This is however not so helpful because the display of those contents are static, or as plain information. Static in the sense that we would still have to read and retrieve the desired summary and/ or information from specific combinations of rows-columns by hand.

With a bit of work we can transition from plain text to live text. And here is where the proper automatization of the work flow begins.

With live text, or in-line code, we can integrate code chunks into plain text so that through `knit` function `rmarkdown` automatically enhances the plain text with the desired information from the external source material. This can be extremely helpful when writing repetitive reports, for instance Another example is when we want to quickly have a look at progress of a data collection process.

The tricky part with live text is to know exactly what to retrieve from the external source material and in what kind of vector that information is stored. Character (text) and numerical vectors behave differently and have different characteristics.

Let us write our first short paragraph that integrates plain text and live text.

. . .

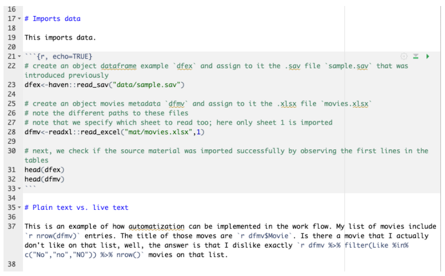
BEGIN EXAMPLE

This is an example of how automatization can be implemented in the work flow. My list of movies include 4 entries. The title of those movies are John Wick, Call me by your name, Terminator, 4 months 3 weeks and 2 days. Is there a movie that I actually don't like on that list, well, the answer is that I dislike exactly 0 movies on that list.

END EXAMPLE

. . .

This is tricky to observe here, so I attach an image of the actual `.Rmd` document.



~~~
# Imports data

This imports data.

```{r, echo=TRUE}
# create an object dataframe example `dfex` and assign to it the .sav file `sample.sav` that was
introduced previously
dfex<-haven::read_sav("data/sample.sav")

# create an object movies metadata `dfmv` and assign to it the .xlsx file `movies.xlsx`
# note the different paths to these files
# note that we specify which sheet to read too; here only sheet 1 is imported
dfmv<-readxl::read_excel("mat/movies.xlsx",1)

# next, we check if the source material was imported successfully by observing the first lines in the
tables
head(dfex)
head(dfmv)
```

# Plain text vs. live text

This is an example of how automatization can be implemented in the work flow. My list of movies include
`r nrow(dfmv)` entries. The title of those moves are `r dfmv$Movie`. Is there a movie that I actually
don't like on that list, well, the answer is that I dislike exactly `r dfmv %>% filter(Like %in%
c("No","no","NO")) %>% nrow()` movies on that list.

~~~

Figure 3.6: Live paragraph in .Rmd

Copy the text from this code chunk into your `.Rmd` file and it should look like in Figure 3.6.

```
This is an example of how automatization can be implemented in the work flow. My list of movie
```

3.0.5 Live text in focus

We've seen that live text can be easily integrated into plain text to create a final document that looks nice. We've also seen that the actual `.Rmd` looks differently because of the in-line coding that supports the automatization of the document.

To write in-line code you only need to follow this simple structure `` `r 1+1` ``. Wrap `r` inside backtrace brackets, add space followed by the code.

If we were to add to the list of movies new entries and we would still want to write that exact paragraph from above then we would simply `knit` the `rmarkdown` document and everything would be automated for us, including the counting of how many entries are in the list, what the title of those movies are and how many movies we actually dislike from that list.

To do this, open the Excel sheet using Microsoft Excel. Type in one or two new entries following the given structure. Save the Excel and then return to `RStudio` and `knit` the `.Rmd` file. You will notice how in the final document the newly added entries in the movie list are now part of the paragraph.

Tip 8: Watch out for structure

This automatization only works if the structure of the external source material remains unchanged with updates.

3.0.6 Automated graphs and tables

One other benefit of working with automated reports is that tables and graphs are automatically updated with new data. This is rather straightforward - one needs to identify the preferred table format and graph layout, and integrate it in a report. With updated data, the report will automatically update contents of the created table and graphs.

To illustrate this, we work with sub-samples from the Stanciu et al. (2017) dataset.

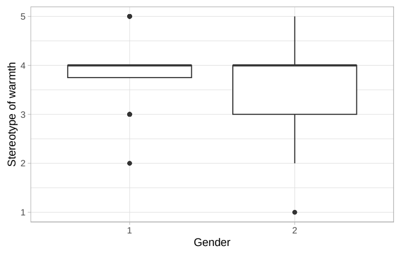
Remember that we have assigned the `sample.sav` dataset as `dfex` dataframe in *R* previously. This dataframe contains information from $N = 100$ study participants. See the dataset description in Chapter 1.

We subset the `dfex` dataframe into a much smaller dataframe `dfex_1` containing $n = 15$ study participants and a somewhat larger dataframe `dfex_2` containing $n = 60$ study participants.

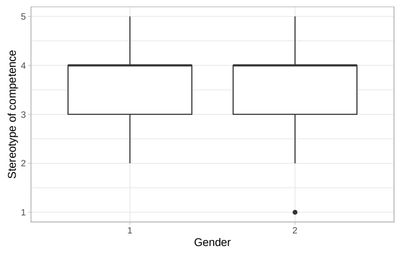
```r
# we remove all the labels using the package sjlabelled and mutate
# as factors columns gen and res
# this step makes it easier later on to work with plots
# note that it is not a necessary step in general but only for the sake of
# simplicity here
dfex<-dfex %>% sjlabelled::remove_all_labels() %>%
  mutate(gen=factor(gen),
         res=factor(res))

# subsamples 15 study participants at random
tmpdf1<-sample_n(dfex,15)
# subsample 60 study participants at random
tmpdf2<-sample_n(dfex,60)
```

3.0.6.1 Graphs

We code a simple plot using the package `ggplot2` from `tidyverse` and submit the three dataframes to the code.



(a) warmth



(b) competence

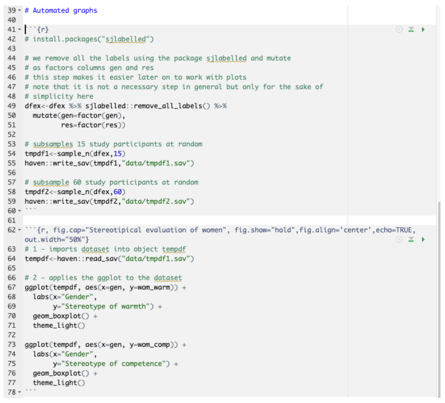
Figure 3.7: Stereotype

Let us now create this imaginary scenario. We save the two smaller `dfex` on the local machine as `.sav` datasets. This step is usually not necessary when the dataset you want to use gets updated by itself.

```r
haven::write_sav(tmpdf1,"data/tmpdf1.sav")
haven::write_sav(tmpdf2,"data/tmpdf2.sav")
```

Now, we're going to pretend that `tmpdf1`, `tmpdf2` and `dfex` are all progressive datasets, meaning that the sample size N increases from 15 in `tmpdf1` to 100 in `dfex`. We're then going to ask for the sample graphs as above in each of the three instances. I won't cover this step here in detail but this can be easily done independently using this small twist.

```r
# import dataset into one object and then subject this object to the ggplot code

# 1 - imports dataset into object tempdf
tempdf<-haven::read_sav("data/tmpdf1.sav")

# 2 - applies the ggplot to the dataset
ggplot(tempdf, aes(x=gen, y=wom_warm)) +
  labs(x="Gender",
       y="Stereotype of warmth") +
  geom_boxplot() +
  theme_light()

ggplot(tempdf, aes(x=gen, y=wom_comp)) +
  labs(x="Gender",
       y="Stereotype of competence") +
  geom_boxplot() +
  theme_light()


# 3 - for illustration purposes, repeat step 1 with each
# of the three datasets (tmpdf1,tmpdf2 and dfex)
# making sure they are assigned into the same object tempdf.
# As long as the ggplot code is applied to a dataset with the same structure
# and variable labels the output will be updated automatically.
```



~~~
# Automated graphs

```{r}
# install.packages("sjlabelled")

# we remove all the labels using the package sjlabelled and mutate
# as factors columns gen and res
# this step makes it easier later on to work with plots
# note that it is not a necessary step in general but only for the sake of
# simplicity here
dfex<-dfex %>% sjlabelled::remove_all_labels() %>%
  mutate(gen=factor(gen),
         res=factor(res))

# subsamples 15 study participants at random
tmpdf1<-sample_n(dfex,15)
haven::write_sav(tmpdf1,"data/tmpdf1.sav")

# subsample 60 study participants at random
tmpdf2<-sample_n(dfex,60)
haven::write_sav(tmpdf2,"data/tmpdf2.sav")
```

```{r, fig.cap="Stereotipical evaluation of women", fig.show="hold",fig.align='center',echo=TRUE,
out.width="50%"}
# 1 - imports dataset into object tempdf
tempdf<-haven::read_sav("data/tmpdf1.sav")

# 2 - applies the ggplot to the dataset
ggplot(tempdf, aes(x=gen, y=wom_warm)) +
  labs(x="Gender",
       y="Stereotype of warmth") +
  geom_boxplot() +
  theme_light()

ggplot(tempdf, aes(x=gen, y=wom_comp)) +
  labs(x="Gender",
       y="Stereotype of competence") +
  geom_boxplot() +
  theme_light()
```
~~~

Figure 3.8: Automated graphs

If all went well, your `.Rmd` would look similar to mine (see Figure 3.8).

3.0.6.2 Tables

You can `knit` a table to your document using `knitr`, `kable` and/ or `kableExtra` packages. Note that there can be differences in whether or not a package returns the desired table layout depending on whether the final `knit`-ed document is in PDF or HTML format. For the sake of simplicity, we only focus in this short book on final documents in HTML format.

```
dfmv %>% knitr::kable(caption="Simple table using knitr::kable()",format = "pipe")
```

Table 3.1: Simple table using knitr::kable()

Movie	Actor	Like	Why	Grade	Wikilink
John Wick	Keanu Reeves	Yes	Fight scenes	10	https://en.wikipedia.org/wiki/John_Wick_(film)
Call me by your name	Timothee Chalamet	Yes	Beautiful love story	10	https://en.wikipedia.org/wiki/Call_Me_by_Your_Name_(film)
Terminator	Arnold Schwarzenegger	Yes	Arnold	9	https://en.wikipedia.org/wiki/The_Terminator

Movie	Actor	Like	Why	Grade	Wikilink
4 months 3 weeks and 2 days	NA	Yes	Portrayal of life in communist Romania	8	https://en.wikipedia.org/wiki/4_Months%2C_3_Weeks_and_2_Da

This is a simple task: Import Excel tables in *R* and then integrate the contents into a final output document. But, imagine you'd want to manipulate somehow the contents of the source material table and create your own table that can be automatically updated with new input in the source material table.

For instance, you might want to create a table of all the movies listed in the source material table where an actor you admire appears in addition to your least liked actor. Say, *Keanu Reeves* is a liked actor whereas *Alec Baldwin* might be a least liked actor.

```
# does some data manipulation to retrieve the required information
tmptbl<-dfmv %>%
  filter(Actor %in% c("Keanu Reeves", "Alec Baldwin"))

# creates an empty table holder that is our summary table that we'd
# want to include in the final output document
extbl<-tibble(

  like=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Like,
  name=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Actor,
  movie=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Movie,
  wiki=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Wikilink

)
```

Now we `knit` the table to the final document. Note that in this particular case no movie by actor Alec Baldwin was listed in the external source material.

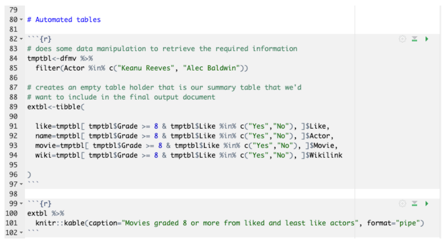
```r
extbl %>% knitr::kable(caption="Movies graded 8 or more from liked and least like actors", for
```

Table 3.2: Movies graded 8 or more from liked and least like actors

like	name	movie	wiki
Yes	Keanu Reeves	John Wick	https://en.wikipedia.org/wiki/John_Wick_(film)

Open Microsoft Excel `movies.xlsx` and add one or more movies by actor **Alec Baldwin** while pretending you dislike the author. Or, you modify the code above and replace the two actors with actors you dislike and like and update the Excel sheet accordingly making sure you maintain the sheet structure.

Then run the code and you should be able to see updated tables now. The code should like something like in Figure 3.9.



~~~
# Automated tables

```{r}
# does some data manipulation to retrieve the required information
tmptbl<-dfmv %>%
  filter(Actor %in% c("Keanu Reeves", "Alec Baldwin"))

# creates an empty table holder that is our summary table that we'd
# want to include in the final output document
extbl<-tibble(

  like=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Like,
  name=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Actor,
  movie=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Movie,
  wiki=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Wikilink

)
```

```{r}
extbl %>%
  knitr::kable(caption="Movies graded 8 or more from liked and least like actors", format="pipe")
```
~~~

Figure 3.9: Code for tables in .Rmd

Knit with parameters

One way to simplify even more the tasks in automatization of the workflow is to use parameters in `knit`-ing a final document. More on working with parameters, and how to publish parameterized reports, can be read here.

Parameters are characteristics of the document that are repetitive both throughout the document and along the iteration of various versions of the document.

Say, you'd want to automatize the writing of a report in each year so the year is a parameter of the report because data, text and tables will have to refer to the in-focus year and thus update the document accordingly.

Say, you'd want to automatize the analysis of data in ways that tables and graphs are identical but for the grouping variable and year of data collection. Grouping variable and year of data publication are parameters of the document because they repeatedly appear throughout the code.

What makes working with parameters useful is the dynamic and user interface this approach brings to automatization of the work flow. Imagine that you'd want colleagues or superiors to easily have access to repeated reports but they do not posses the coding skills required. You can create a parameterized report for them and they can use a simple user interface (`shiny` interface that will be covered in details in Chapter 5) to retrieve the documents they are interested in.

3.0.1 Example progression

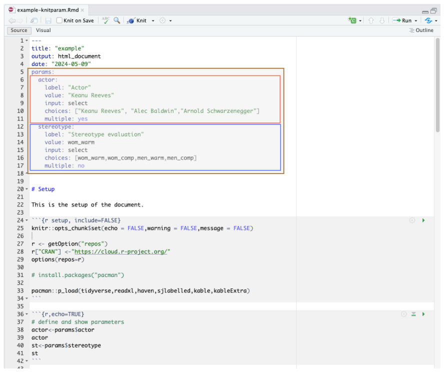
We can transform parts of the `.Rmd` example into parameters and then `knit` the final document using a user interface.

An intuitive parameter is the name of actors in the Excel sheet `movies.xlsx`. We have seen that actors Keanu Reeves and Alec Baldwin are liked and not so liked but, most importantly, we have seen that if one adds entries to that external source material the table will be updated. But, now imagine that we want to personalize that list of movies with our very own liked and disliked actors.

We could also parameterize which of the stereotype evaluation we'd want to use for graph creation. Remember there were four such variables in the `sample.sav` dataset, two each for men and women and each gender was evaluated in view of warmth and competence.

3.0.2 Changing setup to parameterized report

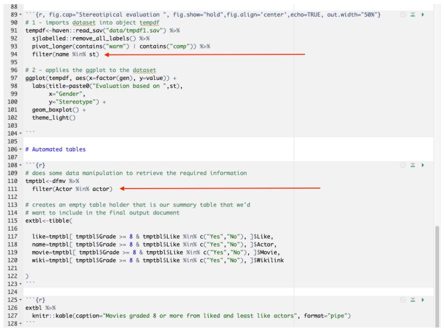
To work with parameters, we first need to define what these are in the `yaml` of the `rmarkdown` document.



````
1  ---
2  title: "example"
3  output: html_document
4  date: "2024-05-09"
5  params:
6    actor:
7      label: "Actor"
8      value: "Keanu Reeves"
9      input: select
10     choices: ["Keanu Reeves", "Alec Baldwin","Arnold Schwarzenegger"]
11     multiple: yes
12   stereotype:
13     label: "Stereotype evaluation"
14     value: wom_warm
15     input: select
16     choices: [wom_warm,wom_comp,men_warm,men_comp]
17     multiple: no
18 
19 
20 # Setup
21 
22 This is the setup of the document.
23 
24 ```{r setup, include=FALSE}
25 knitr::opts_chunk$set(echo = FALSE,warning = FALSE,message = FALSE)
26 
27 r <- getOption("repos")
28 r["CRAN"] <-"https://cloud.r-project.org/"
29 options(repos=r)
30 
31 # install.packages("pacman")
32 
33 pacman::p_load(tidyverse,readxl,haven,sjlabelled,kable,kableExtra)
34 ```
35 
36 ```{r,echo=TRUE}
37 # define and show parameters
38 actor<-params$actor
39 actor
40 st<-params$stereotype
41 st
42 ```
````

Figure 3.10: Modified yaml header for paramtereized reports in .Rmd

In Figure 3.10, I highlighted brownish the modification from the previous `.Rmd` file. Throughout the code lines 5–18, we've added two parameters (`actor` and `stereotype`) to the `rmarkdown` document, which are introduced by the `yaml` attribute `params:`. These parameters are assigned to objects that can be used in `r` as seen in the code chunk at lines 36–42.



~~~

```{r, fig.cap="Stereotipical evaluation ", fig.show="hold",fig.align='center',echo=TRUE, out.width="50%"}
# 1 - imports dataset into object tempdf
tempdf<-haven::read_sav("data/tmpdf1.sav") %>%
  sjlabelled::remove_all_labels() %>%
  pivot_longer(contains("warm") | contains("comp")) %>%
  filter(name %in% st)

# 2 - applies the ggplot to the dataset
ggplot(tempdf, aes(x=factor(gen), y=value)) +
  labs(title=paste0("Evaluation based on ",st),
       x="Gender",
       y="Stereotype") +
  geom_boxplot() +
  theme_light()

```

Automated tables

```{r}
# does some data manipulation to retrieve the required information
tmptbl<-dfmv %>%
  filter(Actor %in% actor)

# creates an empty table holder that is our summary table that we'd
# want to include in the final output document
extbl<-tibble(

  like=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Like,
  name=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Actor,
  movie=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Movie,
  wiki=tmptbl[ tmptbl$Grade >= 8 & tmptbl$Like %in% c("Yes","No"), ]$Wikilink

)
```

```{r}
extbl %>%
  knitr::kable(caption="Movies graded 8 or more from liked and least like actors", format="pipe")
```
~~~

Figure 3.11: Modified code for paramterized reports in .Rmd modified

To accommodate parameters in the previously written code, we need to make some small modifications to the code as seen in

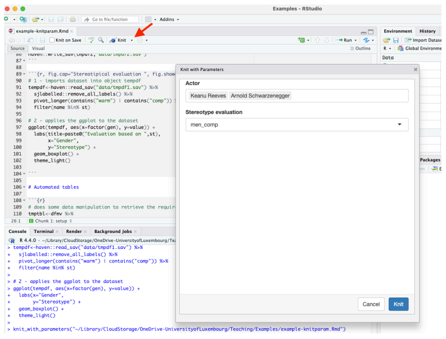
Figure 3.11. Red arrows point to the exact location of parameters in the modified code.

Tip 9: Dataframe modification

Note in Figure 3.10 (or Figure 3.11 for that matter) that we've slightly modified the dataframe so that it gets easier to pass it through parameterization. See lines 91–93 where we've modified the data structure to long format.

3.0.3 Knitting the document

`knit`-ing a document with parameters is as simple as 1-2-3.



```
 87 ```
 88
 89 ```{r, fig.cap="Stereotipical evaluation ", fig.show
 90 # 1 - imports dataset into object tempdf
 91 tempdf<-haven::read_sav("data/tmpdf1.sav") %>%
 92   sjlabelled::remove_all_labels() %>%
 93   pivot_longer(contains("warm") | contains("comp")) 
 94   filter(name %in% st)
 95
 96 # 2 - applies the ggplot to the dataset
 97 ggplot(tempdf, aes(x=factor(gen), y=value)) +
 98   labs(title=paste0("Evaluation based on ",st),
 99        x="Gender",
100        y="Stereotype") +
101   geom_boxplot() +
102   theme_light()
103
104 ```
105
106 # Automated tables
107
108 ```{r}
109 # does some data manipulation to retrieve the requir
110 tmptbl<-dfmv %>%
```

Actor: Keanu Reeves, Arnold Schwarzenegger

Stereotype evaluation: men_comp

```
R 4.4.0 · ~/Library/CloudStorage/OneDrive-UniversityofLuxembourg/Tea
> tempdf<-haven::read_sav("data/tmpdf1.sav") %>%
+   sjlabelled::remove_all_labels() %>%
+   pivot_longer(contains("warm") | contains("comp")) %>%
+   filter(name %in% st)
>
> # 2 - applies the ggplot to the dataset
> ggplot(tempdf, aes(x=factor(gen), y=value)) +
+   labs(x="Gender",
+        y="Stereotype") +
+   geom_boxplot() +
+   theme_light()
>
> knit_with_parameters("~/Library/CloudStorage/OneDrive-UniversityofLuxembourg/Teaching/Examples/example-knitparam.Rmd")
```

Figure 3.12: Knit with parameters

Once the parameterized `rmarkdown` is built, we can `knit` it with parameters as show in Figure 3.12.

Nagvigate to the `knit` button as indicated by the red arrow, open the drop-down menu and from there select "knit with parameters" and following, a window as indicated in Figure 3.12 will appear.

We recognize the two parameters we set up above in the `yaml` header and used throughout the code: "Actor" and "Stereotype evaluation". Note that these are changeble parameter traits meaning that in the `yaml` header we can use the attribute `label` to re-label as per our preference.

We have set the attribute `input: select` in both cases to indicate how the parameter behaves. With `input: select`, we indicate a list of choices to the parameter from which we can then select only one (case of `stereotype`) or multiple (case of `actor`).

3.0.4 Something to do by yourself

Going back to the “Knit with Parameters” window, all the choices we have pre-programmed will appear here. This also mean that if we want to add something, we can simply add it to the `yaml` header and the `r` code will automatically recognize it. Beautiful, clean, and easy.

Play with the “Knit with Parameters” a bit. You could, for example, add to the list of actors **Timothee Chalamet** while deleting one of the other actors. See what happens in the window and in the `knit`-ed output document.

To make things a bit more advanced, open the Microsoft Excel file and add new entries making sure you follow the preset format. Update as desired the choice list in the `yaml` header and then knit using parameters. What output do you get?

3.0.5 Another table example

One other way to work with parameterized reports is to code the document such that it creates tables (or anything else for that matter) using a specific dataset. Basically, if the dataset format is identical but contains different N sizes or was collected by different teams or at different times, then parameterized reports can facilitate the creation of repeated reports at a button’s click.

Remember, we first need to set up the new parameter in the `yaml` header. Try it yourself following the steps above and then use the code below.

```r
# we assign the parameter sampledf to an object sampledf
# containing the dataset itself
# we use paste0() function to integrate the parameter into a string object
# that tells r where to find the dataset in the .Rproj (the file path)
sampledf<-paste0("data/",params$sampledf)
```

```r
sampledf

abc<-haven::read_sav(sampledf)
head(abc) # shows the first lines in the data frame
nrow(abc) # shows the n size of the data frame
```

```r
# assign parameterized data to an object dataframe
abc %>%
  sjlabelled::remove_all_labels() %>%
  pivot_longer(contains("warm") | contains("comp")) %>%
  group_by(name) %>% # we group by variable name created previously (background step)
  summarise(mean=mean(value, na.rm = TRUE), # we use missing remove (na.rm)
            sd=sd(value, na.rm = TRUE),
            min=min(value, na.rm = TRUE),
            max=max(value, na.rm = TRUE))
```

Advanced resources

In this day and age, technology evolves with mind-blowing speeds. This makes it hard to always keep up. Such happens also with the universe of tools available in `r`.

3.0.1 Towards shiny apps

Paramterized reports use a shiny user interface. We will cover `shiny apps` in Chapter 5. But for now be as creative as possible.

For example, think of a nice graph you could build plotting the age of study participants and their stereotypical evaluation of men and women in terms of warmth and competence. Use the `sample.sav` or the shorter datasets we created previously.

Create a new parameter `age` having the `input: slider` attribute. An example of how to do it is given here. Either follow that example or try doing it by yourself.

3.0.2 Quarto

Building on `rmakrdown` et co., the relative `quarto` is making everything much, much easier. Once the basics of `rmarkdown` are secured, the transition to `quarto` is extremely smooth.[2]

What is `quarto` and what makes it so advantageous? See for yourself here.

Use `quarto` to create presentation slides using the `revealjs` format. Start from here.

Tip 10: Presentations in Quarto

Build on the code we've covered so far. Write text, dynamic text from data, and incorporate images in your presentation slides created in Quarto.

[2] We cover `quarto` begining with Chapter 4.

4 Self-publish

Publishing content online is very easy today. One can use WordPress and similar platforms to create a website, blogs and so on.

Why, then, use `r` to publish content online?

Through *R* we can create websites and online books that we can store on online repositories like `GitHub` and maintain a high degree of control over the structure, contents, and visibility of the website/ book. Moreover, self-publishing using `r` is one way to integrate varying work-routines towards a greater goal – that of communicating own research, consultancy job involving data science, to name just a few examples.

If you work with data that comes from your data collection efforts or you use secondary data (publically available data), then self-publishing content online through `r` can facilitate the integration of data analysis and content creation steps.

For this section, we will use `quarto` to create both websites and books. Using `quarto` is a step forward from other approaches (`quarto` is a new generation RMarkdown document), including the way the present book was rendered - using the old generation RMarkdown files.

Prepping Quarto

`Quarto` is a new generation `RMarkdown`. It retains all the functions of the RMarkdown files from before while it enhances and simplifies several other work flows. Basically, `Quarto` makes it much much much easier for everyone to create presentations, live documents, websites, books and so on.

One can even integrate `shiny` functionality into `Quarto` documents. For the present seminar, however, we will introduce `shiny apps` as separate tools (see Chapter 5).

See the official `quarto` website here.

To create content using `quarto` we need:

1 - `r`, which we download and install as described in Chapter 2.

2 - `RStudio`, which we download and install as described in Chapter 2.

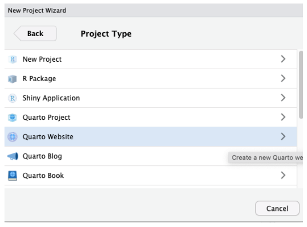
3 - `Quarto`, which can be downloaded here and installed following the indicated steps. Make sure you choose the installation package that suits your operating system.

After you've successfully installed `r`, `RStudio`, and `quarto` on your computer, we can start with creating websites and books.

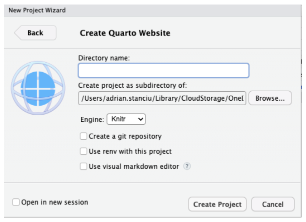
Website

In Chapter 3, we've seen that working in `Rproj` simplifies the work flow including, for example, presetting path dependencies relative to the project folder.

We will use the logic of projects here (and for books) as well. Open `RStudio` and create a new project `Quarto website`. See Figure 4.1.



(a) Step 1



Create Quarto Website

Directory name:

Create project as subdirectory of: /Users/adrian.stanciu/Library/CloudStorage/OneI

Engine: Knitr

☐ Create a git repository

☐ Use renv with this project

☐ Use visual markdown editor

(b) Step 2

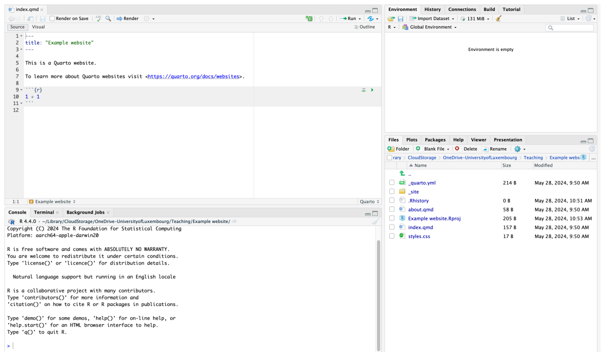
Figure 4.1: Steps to creating a Quarto Website project

When you create the project, there will be several files created by default: `.qmd`, `.yml`, `.css` as well as a folder "_site". The files and folder already contain the fundamental structure of a working website.

To open this default website on your local computer, navigate inside the folder "_site" and open on your Internet browser the `.html`[1] file `index.html`.

Every website (and online book) has such an `index.html` file. This file indexes the other files composing the website (or book).

4.0.1 .qmd



~~~
---
title: "Example website"
---

This is a Quarto website.

To learn more about Quarto websites visit <https://quarto.org/docs/websites>.

```{r}
1 + 1
```
~~~

```
R 4.4.0 · ~/Library/CloudStorage/OneDrive-UniversityofLuxembourg/Teaching/Example website/
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20

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

  Natural language support but running in an English locale

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

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

>
```

rary > CloudStorage > OneDrive-UniversityofLuxembourg > Teaching > Example webs

Name	Size	Modified
..		
_quarto.yml	214 B	May 28, 2024, 9:50 AM
_site		
.Rhistory	0 B	May 28, 2024, 10:51 AM
about.qmd	58 B	May 28, 2024, 9:50 AM
Example website.Rproj	205 B	May 28, 2024, 10:53 AM
index.qmd	157 B	May 28, 2024, 9:50 AM
styles.css	17 B	May 28, 2024, 9:50 AM

Figure 4.2: Basic structure of a .qmd file

[1] `.html` files are specific types of file that are used online

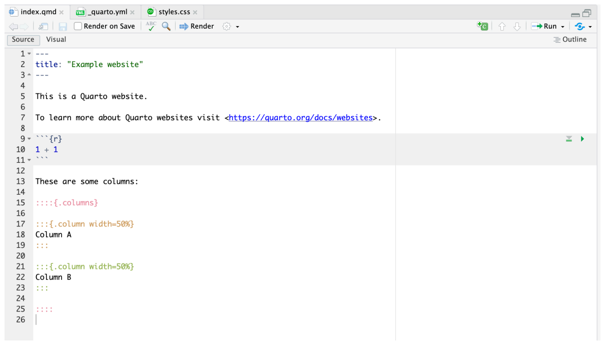
We can see the by now familiar structure including the yaml header at lines 1-3, the r code chunk at lines 9-11 and the static text.

Tip 11: Quarto markdown

`.qmd` is the new generation `.Rmd`

We can also see the `Render` button on the upper bar where the `knit` button would be in an old generation RMarkdown file.

We can edit this file in virtually identical manner to any other `.Rmd` file. I say virtually identically because the `.qmd` file comes with extra benefits making it easier to control the formatting of content on the page. One can easily implement some `.html` features such as `div` using the `:::{}` blocks[2]. Read more on the basics of `.qmd` here.



```
1  ---
2  title: "Example website"
3  ---
4
5  This is a Quarto website.
6
7  To learn more about Quarto websites visit <https://quarto.org/docs/websites>.
8
9  ```{r}
10 1 + 1
11 ```
12
13 These are some columns:
14
15 ::::{.columns}
16
17 :::{.column width=50%}
18 Column A
19 :::
20
21 :::{.column width=50%}
22 Column B
23 :::
24
25 ::::
26
```

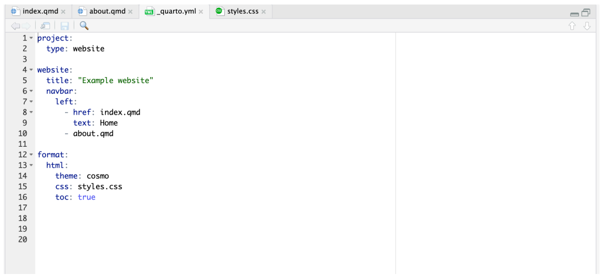
Figure 4.3: Two columns using div blocks

See for example how a two columned format looks like in Figure 4.3. Include it in your `.qmd` file and then render it. See how it looks.

4.0.2 .yml

Three elements are observable when opening the default `.yml` file: `project`, `website`, and `format`. See Figure 4.4.

[2]Some div blocks require four : instead of three, for example ::::{}. Additional : indicate therefore upper levels of div code embededdness.



```
1  project:
2    type: website
3
4  website:
5    title: "Example website"
6    navbar:
7      left:
8        - href: index.qmd
9          text: Home
10       - about.qmd
11
12 format:
13   html:
14     theme: cosmo
15     css: styles.css
16     toc: true
17
18
19
20
```

Figure 4.4: Elements of the .yml file

First, the element `project` defines the type of the project you work one. It can be a website as well as a book.

Second, the element `website` defines attributes of the website. In Figure 4.4 we can see attributes "title" and "navbar". `title` allows us to give a title to the website. `navbar` is short for navigation bar. Let us look a bit closer at it.

Inside `navbar` there is an element `left` which defines the positioning of the navigation bar. It can also be right, for example. On the `navbar` we then place the individual pages - which are none other the `html` renditions of the `.qmd` files. We see two ways to place these individual pages.

One is where we give a custom label to the page: see lines 8 and 9. Another where we use the default label that derives from the title of the individual page (remember that each RMarkdown file has a yaml header and so has every .qmd file).

Tip 12: Listing all the individual pages

Remember to list in the `navbar` all the individual pages you'd like to be rendered in the final website. Otherwise, the website will not contain them!

Third, the element `format` allows us to format the website overall including, for example, a theme, css (cascading style sheets) or even add a table of contents (toc). `theme`, `css` and `toc` and attributes of the `html` document. Remember from Chapter 3 the concept of parameters? Well, that is pretty much the logic here as well - these three are parameters of the html file that we can modify according to our needs. This means that there are multiple themes we can choose from and we can write our very own `.css` style. We may decide to include a `toc` or not.

Tip 13: Further html themes

Select from this list of quarto html themes the one you'd prefer

4.0.3 .css

The default `.css` file is empty. I would say that css is rather advanced so I won't cover it here. Typically, one can create custom css styles for their website or can download templates from the Internet. Be aware however what you download and from where.

For the purpose of this seminar the default `.css` is more than enough.

4.0.4 _site folder

This folder is created by default. It is the containing folder of the rendered html files and other elements required for the final website.

For the purpose of this seminar, this folder is important for us because it contains the `index.html` file. By opening it on our browser, we can inspect locally the rendered website.

4.0.5 Deployment

Once you've created and locally inspected your website, the next step is to deploy it (publish) it online. Remember from Chapter 1 that we would need an online repository like `GitHub` and, most importantly, to have an open and encrypted channel connecting the local machine and the server.

One other way to publish your site is through quartopub.com. Open an account on quartopub.com. We will deploy the website using these two ways: `GitHub` and `quartopub.com`. I will describe a bit why you might want to choose one or the other.

4.0.5.1 Via quartopub

This is a very simple way to deploy your website online. It is integrated seamlessly with `quarto` so you only need two lines of code in the Terminal. Really.

First, we need to render the website. To link together all the files that we've created/ edited included the `.qmd`, `.yml` and `.css`.

Copy the line below into the Terminal of your website project and run it.

```
quarto render
```

Next, we can publish it via quartopub provided that we have opened an account.

Copy the line below into the Terminal of your website project and run it.

```
quarto publish quarto-pub
```

You will notice some code running and finally that in the project folder there is a new yml file created `_publish.yml`. If you open it, you will notice among other the URL to your newly and publicly available website.

With this approach you deploy the website on a remote server and your files remain available for editing only locally on your machine. You also need to make sure that you render the website after each modification to files making up the website.

The created example is published through `quarto pub` here.

4.0.5.2 Via GitHub Pages

In some situations you might want to deploy the website and upload the files comprising the website on a remote repository. This means that your files can be made publicly available for other to clone them or they are available to you in the "cloud". The website itself is deployed and everybody can access it but in addition everyone can have access to the files that create the website.

This approach might be preferable when you maintain a website as a group. Or when you create a fancy website together with colleagues from across the globe.

I wrote a step-by-step guide explanation for older generation RMarkdown. This can be accessed here.

Returning to Quarto Website and assuming that a `GitHub` account has been created, you may follow the steps here. A summary of the steps are provided below:

1 - Create a `Quarto Website` project on your local machine.

2 - Make the folder containing the `quarto` website a `git` repository[3].

```
git init
```

3 - Create an empty repository on your `GitHub` account. Then, connect the local repository to the newly created online repository.

```
remote add origin git@github.com:{your github user}/{your repository where the website will be
```

4 - We will deploy the website by rendering it to a sub-folder docs. To do this we need to modify the `_quarto.yml` document. Open the `.yml` file, copy the following line and paste it as a sub-element to "project". This new line should be intended and be aligned with the "type: website" element! Save and close the `.yml` file.

[3]In Chapter 1 we covered the set up of GitHub and repositories including installing a suite of code. Make sure that is done before attempting to create the folder a `git` repository.

```
output-dir: docs
```

5 - Create a `.nojekyll` file to the repository. Explanation is given here. Go to the Terminal of the website project and run the code line:

Mac/ Linux

```
touch .nojekyll
```

Windows

```
copy NUL .nojekyll
```

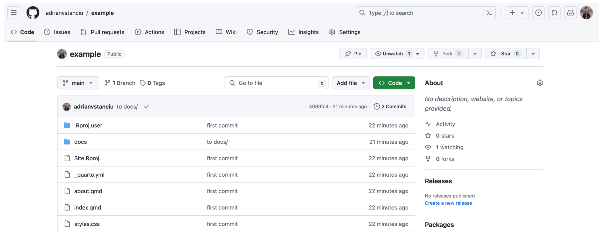
6 - Render the website. Note that the website will be automatically rendered in the sub-folder `docs`.

```
quarto render
```

7 - We push everything to our `GitHub` repository.

```
git add .
git commit -m "Push website"
git push
```

8 - Check that the push was successful.



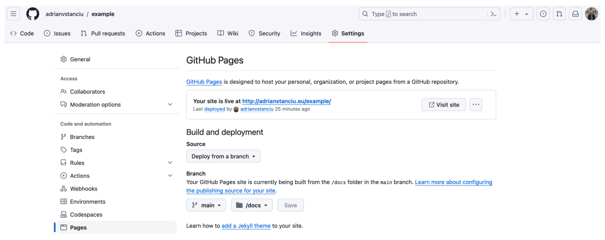
adrianvstanciu to docs/ 4089fc4 · 21 minutes ago 2 Commits

No description, website, or topics provided.

.Rproj.user	first commit	22 minutes ago
docs	to docs/	21 minutes ago
Site.Rproj	first commit	22 minutes ago
_quarto.yml	first commit	22 minutes ago
about.qmd	first commit	22 minutes ago
index.qmd	first commit	22 minutes ago
styles.css	first commit	22 minutes ago

Figure 4.5: Contents of the created GitHub repository

Go on your `GitHub` account, open the repository and it should look something like Figure 4.5.



GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

Your site is live at http://adrianstanciu.eu/example/
Last deployed by adrianvstanciu 25 minutes ago

Build and deployment

Source
Deploy from a branch

Branch
Your GitHub Pages site is currently being built from the `/docs` folder in the `main` branch. Learn more about configuring the publishing source for your site.

main /docs

Learn how to add a Jekyll theme to your site.

Figure 4.6: Setup for deployment from main/docs

9 - Setup `GitHub` to "read" the website from the sub-folder `docs`.

Navigate to Settings/ Pages and change under "Build and deployment" the branch and folder according to Figure 4.6. Save changes.

10 - A visit site panel will appear shortly, after the website has been rendered and created, like in Figure 4.6. This may take between a few seconds and a couple of minutes depending on the server availability. Refresh the page and the website is now publicly accessible.

11 - Now you can repeat steps 6 through 8 every time you edited or modified contents for your website. The website itself will be updated automatically (might take a couple of minutes though).

The created example is published via `GitHub pages` here[4].

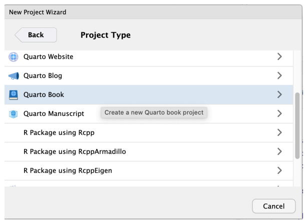
Tip 14: Custom domains

You can publish your website (or book, as we will see in the next section) on a custom domain. For this you'd need to pay a monthly or yearly fee.
See for details this ultra-brief guide I wrote some time ago but still remains valid.

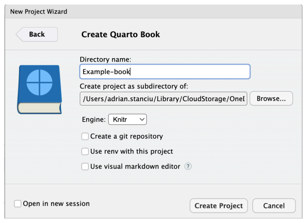
[4]Note that I am using a custom domain at the root of which is my personal website. In your case it would have a slightly different domain but the website itself will look similar to the example shown.

Online book

To start with Quarto Books, we should first create such a project. The steps are similar to creating a Quarto Website.[5]



(a) Step 1



Directory name: Example-book
Create project as subdirectory of: /Users/adrian.stanciu/Library/CloudStorage/OneI
Engine: Knitr
Create a git repository
Use renv with this project
Use visual markdown editor
Open in new session

(b) Step 2

Figure 4.7: Steps to creating a Quarto Book project

When you create the project (see Figure 4.7), there will be several files created by default: `.qmd`, `.yml`, `.bib` as well as a folder "_book". The files and folder already contain the fundamental structure of a renderable book.

To open this default book on your local computer, navigate inside the folder "_book" and open on your Internet browser the `index.html`.

4.0.1 .qmd

See sub-section on .qmd for creating websites.

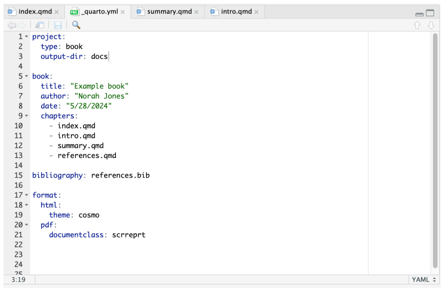
If you feel comfortable, create some content for the book, for example, add a chapter about yourself.

4.0.2 .yml

This file type is identical to the one for creating websites. But, there are different attributes addressed! See Figure 4.8.

First to notice, is that now the project we create is `type: book`. We have learned from the website deployment section that, if

[5]As a matter of fact, you might have noticed by now that creating an R project is this simple. You can follow these steps also when creating for example a presentation or a publishable manuscript.



```yaml
1  project:
2    type: book
3    output-dir: docs
4
5  book:
6    title: "Example book"
7    author: "Norah Jones"
8    date: "5/28/2024"
9    chapters:
10     - index.qmd
11     - intro.qmd
12     - summary.qmd
13     - references.qmd
14
15 bibliography: references.bib
16
17 format:
18   html:
19     theme: cosmo
20   pdf:
21     documentclass: scrreprt
22
23
24
```

Figure 4.8: Attributes of the .yml file for books

we wanted, we could deploy the book via `GitHub Pages`. That is why I already added the `output-dir: docs` in Figure 4.8.

Lines 5 through 13 hold attributes of the book. Remember that for the website, we had specified attributes for website. So, at line 4 in Figure 4.4 we had `website:` followed by attributes whereas at line 5 in Figure 4.8 we have `book:` followed by several attributes.

In our example, we have four book attributes including a title, author, publication date and chapters. For a comprehensive list of book attributes see here.

> Tip 15: List all the chapters!
>
> Remember to list under `chapters:` all the `.qmd` chapters you'd want to render to the final book. Otherwise these are not included. Note also that the order of chapters is defined here.

At line 15, we see the `bibliography:` entry. We will cover this shortly. But, this element is relevant for books and publishable manuscripts alike. It renders to the final document a `.bib` file that contains references.

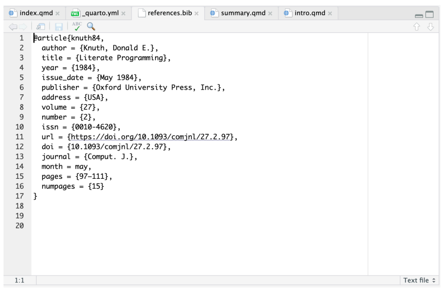
At lines 17 through 21, we see the `format:` element, which is similar to website creation. Next to the familiar `.html` format, we now see also `.pdf` format. This means that we can render the book both in html and pdf formats! Other formats are likewise possible, for instance, `.epub` which are for e-readers.

Tip 16: PDF of your book

You may inspect the PDF version of your book by navigating inside the "_book" (or "docs", depending on the output-directory indicated in the `.yml` file) folder provided, of course, that you first rendered the book using, for example, the code line `quarto render`.

4.0.3 .bib

The `.bib` file format is otherwise said what makes it possible to integrate citations to your book or publishable documents as part of the workflow. `.bib` files are a type of plain text files (no hyperlinks or enhanced fields that are typical in standard text editors like Microsoft Word).



```
@article{knuth84,
  author = {Knuth, Donald E.},
  title = {Literate Programming},
  year = {1984},
  issue_date = {May 1984},
  publisher = {Oxford University Press, Inc.},
  address = {USA},
  volume = {27},
  number = {2},
  issn = {0010-4620},
  url = {https://doi.org/10.1093/comjnl/27.2.97},
  doi = {10.1093/comjnl/27.2.97},
  journal = {Comput. J.},
  month = may,
  pages = {97–111},
  numpages = {15}
}
```

Figure 4.9: Example of reference in `.bib` format

Read more on using `.bib` reference files here.

Many journals offer the possibility to download a `.bib` citation of their articles. For example, navigate to this article Conner and Armitage (1998) published at Journal of Applied Social Psychology (open access). Identify the `TOOLS` button, press it, and navigate to `Export Citation` where you can download a BibTex (`.bib`) citation ready to be incorporated in your reference file.

One can use online tools for transforming plain text citations into `.bib` citation format. For example, this converter. Use this if you have a plain text list of references and you'd want them transferred into a `.bib` file.

So, in our case, as shown in Figure 4.8 at line 15, we specify `bibliography: references.bib` meaning that the bibliography for the book should be constructed through rendering from the file `references.bib`.

4.0.4 Deployment

4.0.4.1 Via quartopub

Book deployment is almost identical with the deployment of websites.

The easiest option is to deploy through `quarto pub`. To do this, we only need to do the following:

1 - Make sure we have opened an account on quarto.com

2 - render the book using the line code below (identical to the one used for websites)

```
quarto render
```

It may take a couple of seconds until the book is rendered. And, if it is the first time you render a qurto book it may take a bit until the necessary dependencies are installed.

3 - Publish the book using one line of code.

```
quarto publish quarto-pub
```

The example book was published online through `quarto.pub` here.

4.0.4.2 Via GitHub Pages

The example book was published online via `GitHub pages` here. Similar to the website example, the book is attributed to my personal website.

Advanced resources

There are some alternatives out there, for example:

- To create a blog using the `blogdown` package. See a step-by-step guide by **Shilaan Alzahawi**.
- To create a website using the `distill` package. See a guide by **Sally A.M. Hogenboom**.
- A more comprenesive guide to creating a website using `r` and `RStudio` is provided in this eBook by **Danny Morris**.
- A different (using older generation RMarkdown) way to self-publish books online using `r` package `bookdown`. See this guide by **Yihui Xie** who basically (co-)created all of this that we are covering throughout the seminar.

Last, but not least. With `quarto`, one can publish even articles. That is right, one can write using `r` publishable manuscripts. Start here and see here the journals that already support manuscript submission of `quarto` templates. This is the cherry on top of the cake in view open and transparent science.

5 Shiny apps

Shiny applications, in short `shiny apps`, are applications created in `r` with a `shiny` user interface. We have seen for the first time the `shiny` user interface in Chapter 3 when we created parameterized reports.

`shiny apps` are extremely useful when the goal is to engage with the audience or readership or supervisors in an interactive and dynamic manner. This application type is powerful because it builds on the programming language `r` and integrates user design features.[1] [2]

Here are examples of `shiny apps` from my work and others.[3]

[1] `python` programming language is likewise supported but won't be covered in this short book. See the official website for more.

[2] One can integrate `shiny app` features in `quarto` documents as described here.

[3] Some of these apps might take a bit of time until they load because they might be "asleep". A shiny app is asleep when there is no activity for a pre-determined time thus the server cleans up working memory by putting inactive apps to sleep.

1 - Predicted as observed created by Dr. Julian Kohnke. Read explanation paper by Witte, Stanciu, and Zenker (2022).

2 - Quantum social sciences created by myself. Read explanation preprint by Witte and Stanciu (2023).

3 - Elements of cross-cultural research created by Maksim Rudnev.

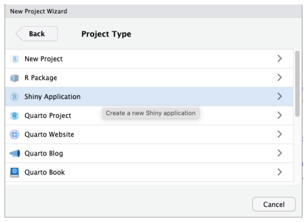
Generally, I feel that the beginner's guide on the official shiny website is more than enough. In this section, we will cover only the basics and address some of the tricky and subtle knowledge about writing code for `shiny apps`. There are not many, but the few that are can become frustrating.

To make things a bit more interesting, we will transform the parameterized report build in Chapter 3 into a shiny app while adding a few additional interactive features to it.

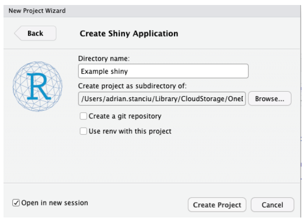
Tip 17: Shiny guide

The official shiny beginner's guide is simple, useful, and full of fun and interactive examples. Navigate to this guide here.

The set-up



(a) Step 1



Directory name: Example shiny
Create project as subdirectory of: /Users/adrian.stanciu/Library/CloudStorage/OneI
☐ Create a git repository
☐ Use renv with this project
☑ Open in new session

(b) Step 2

Figure 5.1: Steps to creating a Shiny Application project

Figure 5.1 should by now be a familiar routine: We create an R project that has a default working app. Once you've created the project, you will note only one file: `app.R` (next to the `.Rproj` extension, of course).

The `.R` file format is an `r` script file format. Meaning, that you can write `r` code, save it and even run it without having to transfer the code to the console. Remember that in RMarkdown and quarto documents, we worked with `r` code chunks. Well, the `.R` script files are much like a gianormous code chunk.

Remember the dataset Stanciu et al. (2017) introduced in Chapter 1? We will use it in creating our very own `shiny app`.

> Tip 18: R script files
>
> Note that `r` script files can hold only basic code meaning that features from RMarkdown and quarto are not available. If you'd like to integrate HTML or other languages, that is possible, but requires appropriate integration which we do not cover here.

If everything went well, we should have the `shiny` package already installed from Chapter 1. But, if you haven't done so yet, or are unsure about it, now is a good time to install this package.

```r
install.packages("shiny")
library(shiny)
```

R script files

`.R` script files can be very useful not only for `shiny apps` but also for websites or books. Writing code in such file format can keep the work environment neat and tidy. Also, you might want to use script files when you know in advance that you have repetitive code (e.g., functions) that you use across several of your projects.

One example where I personally apply this work strategy is when writing my own functions (see Chapter 2). Instead of writing a function each time I would need it, I include all functions I create into an `.R` script file which I then copy-paste to all my projects. Other strategies are possible, for example, install all packages in a separate script file or divide the work flow in programming into several steps - data import, data cleaning, data manipulation, data analysis, and data ready to report.

When working with code stored in separate script files we need to call these into the main document, for instance, in an RMarkdown, quarto or shiny app.

This is easily done with the function `source("{yourrscript here}")`. Make sure that the path dependency is correct and that the script is wrapped around `" "`.

Tip 19: Calling R data files

When we work with data saved in an *R* format such as `.Rds` and `.Rdata`, we can call this dataframe using the function `load("{your .Rdata dataframe here}")`. Remember the path dependencies and the `" "`!

Before we start working on the `shiny app` let us save the data (Stanciu et al. 2017) into an `.Rdata` format. This makes it somewhat easier to import the dataframe in the shiny app code, as it is already in an `r` format.

Building on the code written in Chapter 3, we save to an `.Rdata` file format.

```r
# imports data in SPSS .sav format
dfex<-haven::read_sav("data/sample.sav")

# saves R object dfex into an .Rdata format
# which we load into the shiny app shortly
save(dfex,file="data/sample.Rdata")
```

In Chapter 2 we wrote a basic function to add a constant to all numeric columns to a data frame. We can copy this function into an `.R` script so that we have access to it in writing our first `shiny app`.

Create a new `.R` script file by navigating to File/ New File/ R Script. It will open an empty untitled script file. Copy the function as is into this script file.

```
func1<-function(df,n){

  tmp <- Filter(is.numeric, df) # we first filter the dataframe for numeric columns

  tmp + n # we then add the constant to all the numeric columns
}
```

Tip 20: Packes for `r` scripts

Note that if you write custom scripts and store them inside `r` script files, you'd need to make sure that the required packages are called inside that script file. Use the `install.packages()` or `library()` commands as described in Chapter 2.

Shiny apps structure

What makes `shiny apps` powerful and at the same time a bit tricky to program is the structure. Shiny apps have a user interface (UI) that is wrapped around code that runs in the background on a server. When programming a `shiny app` therefore we need to program both the design (UI) and the code that runs on the server (server).

The UI part makes a `shiny app` attractive to the audience and, if programmed right, can engage the audience in an interactive and dynamic manner. Programming the UI part requires a bit of orientation toward the audience for which the app is designed. What are the minimum skills required to operate the app? What theoretical and practical expertise is expected for the audience to intuitively navigate the app? Read more on user interface in general on the Wikipedia page.

The server part makes a `shiny app`, well, work. Here is where code is written to import, clean, manipulate and analyse data, metadata and all sorts of other things. One way that I find helpful to think of the server part is to see it as the old-school *R* coding on my local machine. When you use `r` for data analysis, for example, you use this programming language in the console which then you run resulting in some form of output. Well, this means technically that you interact with your computational machine (CPU, for example) through the `r` programming language. This very principle applies also for writing code for the server for `shiny apps`.

This distinction is less intuitive when we run the `shiny app` on the local machine. But, this distinction between UI and server becomes crucial when we deploy the app on online repositories, as we will see shortly. By deploying the app code structured into UI and server, we tell the respective servers how to read our code.

So, long story short, both the UI and server segments of a `shiny app` code has its own pre-defined role and it is crucial for the well functioning of the app that this structure is maintained. Otherwise the app breaks.

Code for UI

This will not be a comprehensive code at all. But, it should offer sufficient hands-on tips on how to start building your UI for your first `shiny app`.

One thing to keep on the back of your mind is that in the UI part we need to refer to objects from the server part. If we do not call objects from the server in the UI part properly, the app might still work but the audience will not have access to it.

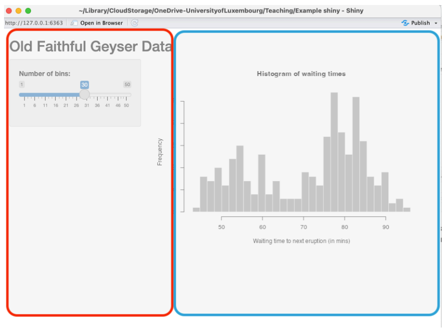
Tip 21: Commas and brackets!

Make sure that you always use commas and close the brackets appropriately. Otherwise, the design might not look as intended or the entire code might break even.

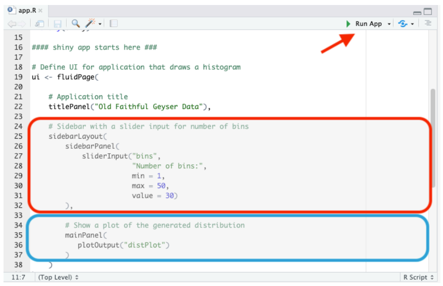
My recommendation is to take some time to decide what do you want to include in the app and what do you need for your audience. For example, do you want the audience to view plots or tables, and if yes, do you want these to be interactive? If that is the case, what code do you need to write on the server part and what is the final `r` object that you'd want to be displayed for the audience via the UI?

So, for me at least, writing a `shiny app` is a bit of a forth and back between the UI and server code.

5.0.1 Layout


~/Library/CloudStorage/OneDrive-UniversityofLuxembourg/Teaching/Example shiny - Shiny
http://127.0.0.1:6363
Open in Browser
Publish
Old Faithful Geyser Data
Number of bins:
Histogram of waiting times
Frequency
Waiting time to next eruption (in mins)

(a) Basic panels of a `shiny app`.



```
#### shiny app starts here ###

# Define UI for application that draws a histogram
ui <- fluidPage(

    # Application title
    titlePanel("Old Faithful Geyser Data"),

    # Sidebar with a slider input for number of bins
    sidebarLayout(
        sidebarPanel(
            sliderInput("bins",
                        "Number of bins:",
                        min = 1,
                        max = 50,
                        value = 30)
        ),

        # Show a plot of the generated distribution
        mainPanel(
           plotOutput("distPlot")
        )
    )
```

(b) Corresponding code in the UI.

Figure 5.2: UI code and corresponding shiny outcome

`sidebarLayout()`: Inside this function we define the content that will be displayed on the side of the app window. See red panel of Figure 5.2a and the corresponding red panel in Figure 5.2b. One can also add elements to the layout of this panel. For example try out the code below and see what happens.

```
# copy and paste this code line as the first argument inside the
# sidebarLayout() function followed by a comma
# I cannot stress this enough: Commas are super super important
# so do not forget them
```

```
position = "right"
```

`sidebarPanel()`: This is wrapped inside the `sidebarLayout()` because it is just one element of several that can be placed on the `sidebarLayout` of the app.

The attributes defined here are fed into the code on the server, so make sure you chose the appropriate user input type.

`mainPanel()`: This contains the output, be it plain text, live text, tables or figures. If in the `sidebarPanel()` you define the user input attributes, in the `mainPanel()` you simply call the objects computed on the server and programm how exactly will they be displayed. See in blue Figure 5.2a and the corresponding code in blue inside the UI Figure 5.2b.

5.0.2 Input types

Inside the `sidebarPanel()` we can define the kind of user input we expect our audience to play with. That is – remember parameterized reports from Chapter 3 – what are the parameters that users can interact with. There are a couple of input types, for example, slider input on a continuous pre-defined numeric range, select from a pre-defined list, check if TRUE or FALSE, and a numeric only text field.

`sliderInput()`: This is given as an example in the default `shiny app` that comes pre-set when creating a shiny app project. It is a slider input type. Observable are five attributes:

- Label of the input `"bins"` which is a reference label for the code
- Label of the input displayed to users (`"Number of bins:"`). Note that this is different than input label above and serves only the function of informing the user. The input label above serves the function of cross-reference in code writing.
- `min` and `max` define the minimum and maximum of the numeric range of the slider.

- `value` defines the default value of the slider which is displayed every time the app is called.

`selectInput()`: This is a select from a pre-defined list input type. In the basic form, it requires three attributes:

- Label of input for cross-referencing in the code.
- Label of input for display for the app users.
- Values of the pre-defined list. Note that these should be places inside a list, which has this basic structure:

```
c("{NAME 1 TO BE DISPLAYED}" = "{value 1 for code cross-referencing}",
"{NAME 2 TO BE DISPLAYED}" = "{value 2 for code cross-referencing}"...)
```

It may look like this (code from Witte and Stanciu (2023)).

```
selectInput("hov",
          "Choice: ",
        c("all",
          "Openness to change" = "och",
          "Conservation" = "con",
          "Self Transcendence" = "str",
          "Self Enhancement" = "sen")
                     )#closes selectInput
```

`checkboxInput()`: This is a yes/ no logical input type. Ideally, you always define at least three atributes:

- Label of input for cross-referencing in the code.
- Label of input for display for the app users.
- `value` is true (`value = T`) or false (`value = F`).

It may look like this (code from Witte and Stanciu (2023)).

```
checkboxInput("p1.ess","Distribution", value = F)
```

`numericInput()`: This an input type that allows the user to type in numeric values within a pre-defined range with a pre-defined increment value. One can define the following:

- Label of input for cross-referencing in the code.
- Label of input for display for the app users.
- `value` is the default value show every time the app is opened.
- `min` and `max` define the range of possible values within which the user can choose to enter from. Note that this range is not visible to the user but it is a by-design-limitation. An error is shown or simply the input is not validated if the user enters a value outside this pre-defined range.
- `step` defines the increment value. It can be a full integral number or anything inbetween.

It may look like this (code from Witte and Stanciu (2023)).

```
numericInput("n",
                                 label = "Sample size",
                                 value = 20,
                                 min = 20,
                                 max = 1000,
                                 step = 1)
```

5.0.3 Conditional panels

There might be situations where you'd want to create a conditional user interface. This means that the UI experience can, at some pre-defined parts, be conditional on user input. For instance, for the app Quantum Social Science, I created a UI dependent on type of analysis: choice. There are three choices the user can select: Simulations, Survey data or Experimental data (which is still under construction). Depending on the user choice at this stage, the user has different options to choose from - either interact with simulated data or with secondary data.

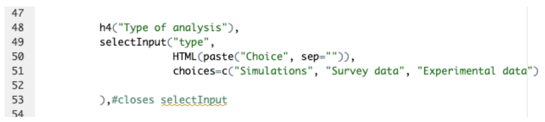
It may look like (code from Witte and Stanciu (2023)).

```
  conditionalPanel(
            condition= "input.type=='Simulations'",
... # the code continues here with input values
```

`conditionalPanel()`: Wrapped inside one can code the UI conditional on an input defined at a previous stage. Let us pay a closer look at the example above:

`condition = "input.type=='Simulations'"`:

- `condition =` introduces the condition that needs to hold for the contents of the rest of conditionalPanel to be activated.
- `"input.type=='Simulations'"` is the condition itself which is to be read as follows: if the input of input object "type" is identical to "Simulations", then the subsequent contents are activated. See Figure 5.3.



```
        h4("Type of analysis"),
        selectInput("type",
                    HTML(paste("Choice", sep="")),
                    choices=c("Simulations", "Survey data", "Experimental data")

        ),#closes selectInput
```

Figure 5.3: Exerpt from UI code of Witte and Stanciu (2023)

5.0.4 Tabset

There can be situations where it is helpful to organize output into separate panels – similar logic to having several tabs open on your web browser.

The app Predicted as observed created by Julian Kohne for the paper Witte, Stanciu, and Zenker (2022) nicely uses this feature.

In the `mainPanel()` the "Abstract" of the paper, followed by "Check assumptions", the calculation of the "Similarity Index" and display of the "Similarity Interval", and, finally, "Recommendations" are organized neatly into tabs. The user can navigate these tabs knowing the kind of content to expect.

`tabsetPanel()`: defines the overall structure within which multiple panels can be placed.

`tabPanel()`: defines the content be placed inside a tab. This is coded inside `tabsetPabel()`!

It may look something like this.

```
tabsetPabel(
  tabPanel1("label 1 for display to user", {content 1 here}),
  tabPanel2("label 2 for display to user", {content 2 here})
)
```

Code for server

The code for the server is a custom function, a gianormous custom function! Like any custom functions (see Chapter 2), there is a structure to it, namely `function(){}`.

The server function takes two arguments, namely `input` and `output`.

`input` signals what comes from the UI interface. That is, what the user of the app is inputing via the UI.

`output` signals what goes from the server to the UI. That is, what the user views as a result of interacting with the app.

5.0.1 Reactive objects

The simplest way to think of reactive objects is to see them as plain old-school *R* code wrapped inside an object that the server needs to compute. It is reactive, because the server has to first compute the reactive object before performing any tasks that call on such an object.

In `r` code written for computation on the local machine it would be called simply an `r` object.

On the server, however, code is computed only when needed which makes objects created on the server reactive to code that requires them. They react if you poke them. Otherwise, they sleep.

Tip 22: Mind the brackets

This is one of those ultra small details that took me days to figure out. When you create a reactive object, remember to always call it as such. It is an `r` object all right, but it looks like a function: `reactiveobject()`.

A reactive object, like all objects coded for the server, need to be wrapped in a specific function, otherwise the server will not recognize it as such.

Tip 23: Server code

Code for the server – whether it is reactive, input or output objects – needs to be written inside `({YOUR SERVER CODE HERE})`. It is a specific code chunk for the server. Its logic is similar to the code chunk introduced in Chapter 3 – it is a field recognized by the server as code to be computed.

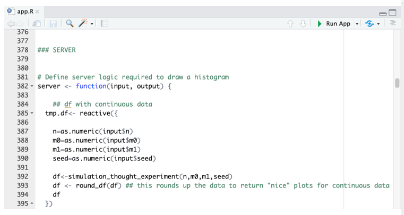
Once a reactive object was coded following the code structure introduce in Tip 23, the reactive object itself can be called inside other code on the server following the structure presented in Tip 22.

5.0.2 Input objects

Technically speaking, input objects are reactive objects. But, I discuss them separately because these feed user input to the code.

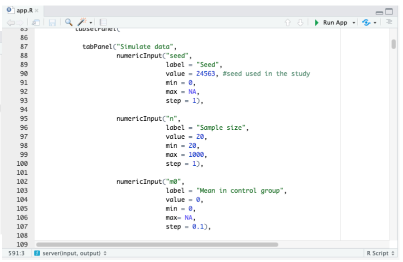
This can be recognized and done by adding the prefix `input$` when calling user input.

In the code example from Witte, Stanciu, and Zenker (2022) (see Figure 5.4a), a reactive object `tmp.df` is coded at lines 385 – 395.



```
376
377
378 ### SERVER
379
380
381 # Define server logic required to draw a histogram
382 server <- function(input, output) {
383
384     ## df with continuous data
385   tmp.df<- reactive({
386
387     n=as.numeric(input$n)
388     m0=as.numeric(input$m0)
389     m1=as.numeric(input$m1)
390     seed=as.numeric(input$seed)
391
392     df<-simulation_thought_experiment(n,m0,m1,seed)
393     df <- round_df(df) ## this rounds up the data to return "nice" plots for continuous data
394     df
395   })
```

(a) Example code reactive and input object.



```
85            tabsetPanel(
86
87              tabPanel("Simulate data",
88                       numericInput("seed",
89                                    label = "Seed",
90                                    value = 24563, #seed used in the study
91                                    min = 0,
92                                    max = NA,
93                                    step = 1),
94
95                       numericInput("n",
96                                    label = "Sample size",
97                                    value = 20,
98                                    min = 20,
99                                    max = 1000,
100                                   step = 1),
101
102                      numericInput("m0",
103                                   label = "Mean in control group",
104                                   value = 0,
105                                   min = 0,
106                                   max= NA,
107                                   step = 0.1),
108
109
591:3  server(input, output)                                          R Script
```

(b) Corresponding UI code for input objects.

Figure 5.4: Code example for reactive objects with user input

This reactive object happens to be using user input information as indicated at lines 387 – 390.

To illustrate the correspondence between UI and the server, see in Figure 5.4b the user defined object `n` at lines 95 – 100 and the user defined object `m0` at lines 102 – 107.

5.0.3 Output objects

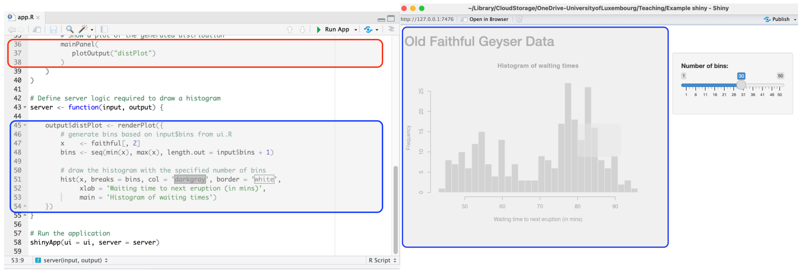
Output objects are the output that we want to be displayed on the user interface. This means that we have to call it as such and indicate the position where we want it displayed.

Ideally, we have already coded the display position and display characteristics in the UI code.

On the server, we need to indicate the object corresponding to the UI code accordingly. When we feed user input to the server code we write the prefix `input${USER INPUT}`. Well, when we want an object displyed on the user interface we write the prefix `output${corresponding label in the UI code}`.

Note the dollar sign `$`!

In Figure 5.5a, we see at lines 36 – 38 (red field) the UI code for object `distPlot` to be displayed. At lines, 45 – 54 (blue field), we see the server code for this object. Finally, in Figure 5.5b (blue field), we see the output plot displayed for user experience.



```r
        # Show a plot of the generated distribution
        mainPanel(
           plotOutput("distPlot")
        )
    )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

    output$distPlot <- renderPlot({
        # generate bins based on input$bins from ui.R
        x    <- faithful[, 2]
        bins <- seq(min(x), max(x), length.out = input$bins + 1)

        # draw the histogram with the specified number of bins
        hist(x, breaks = bins, col = 'darkgray', border = 'white',
             xlab = 'Waiting time to next eruption (in mins)',
             main = 'Histogram of waiting times')
    })
}

# Run the application
shinyApp(ui = ui, server = server)
```

(a) UI code and corresponding server code for output object.

(b) Corresponding plot display in the user interface.

Figure 5.5: Shiny app full circle: ui, server and user experience

Tip 24: Output objects need adequate functions

When writing output objects, these need to be wrapped inside designated code chunks – for plots or tables or text. See this official cheat sheet.

Run the app locally

Running the app locally is as simple as pressing the button `Run App` on the bar – see for instance Figure 5.2b.

Note however that this is in fact calling a function written inside the `app.R` script.

This function is `shinyApp(ui = ui,server = server)`. The `shinyApp()` function takes two arguments `ui` and `server` which we define separately, as indicated above.

A new window will open. You might note two things in the console:

1. The console is busy with the app, as indicated by the STOP symbol.
2. The text in the console Listening on {an IP address}.

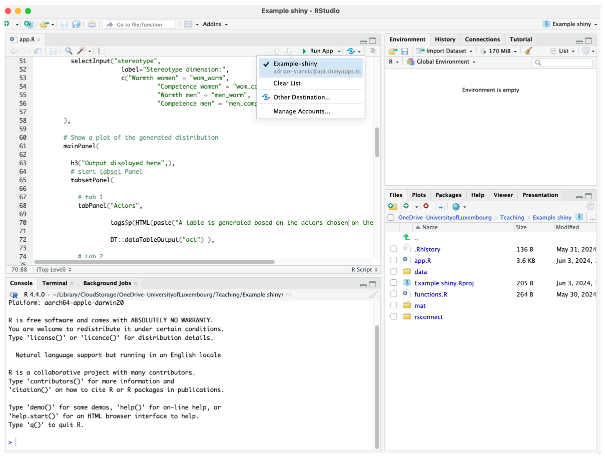
These two things indicate that the app is running and that the console cannot be used for other purposes. It also means that the app is automatically updated if you are to modify the code, UI or server. It also means that your local machine acts as the server in this case.

The moment you close the app, the console becomes available once more.

Deployment

To deploy the app, we would need a dedicated server and, of course, an access account on that server. One efficient and smooth way to deploy a `shiny app` online is to use the dedicated server shinyapps.io. It is a free service maintained by the same community behind `RStudio` – posit.co.

To start with, open an account on shinyapps.io. Once you have an account, we can turn back to our app that we've coded in `RStudio`.



```
          selectInput("stereotype",
                      label="Stereotype dimension:",
                      c("Warmth women" = "wom_warm",
                           "Competence women" = "wom_co
                           "Warmth men" = "men_warm",
                           "Competence men" = "men_comp

        ),

        # Show a plot of the generated distribution
        mainPanel(

          h3("Output displayed here",),
          # start tabset Panel
          tabsetPanel(

            # tab 1
            tabPanel("Actors",

                     tags$p(HTML(paste("A table is generated based on the actors chosen on the

                     DT::dataTableOutput("act") ),

            # tab 2
```

```
R 4.4.0 · ~/Library/CloudStorage/OneDrive-UniversityofLuxembourg/Teaching/Example shiny/
Platform: aarch64-apple-darwin20

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

  Natural language support but running in an English locale

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

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

>
```

Name	Size	Modified
..		
.Rhistory	136 B	May 31, 2024
app.R	3.6 KB	Jun 3, 2024,
data		
Example shiny.Rproj	205 B	Jun 3, 2024,
functions.R	264 B	May 30, 2024
mat		
rsconnect		

Figure 5.6: Steps to `shiny app` deployment on external server.

Next to the button `Run App` there is another button called `Publish the application or document`. Click on it and follow the steps as indicated. Note that in Figure 5.6, the app has already been deployed so there is a connection with the external server made. If you publish an app for the first time, you will only see the option `Publish Website...`. Select from the options `shinyapps.io` and follow the instructions. In a matter of a few secods your app will be online![4]

> Tip 25: Select your files to deploy
>
> Not all files are needed for the final app to function. When you deploy the app, select those files from your local repository that the app actually needs to function. By doing this, you make sure that the server is not filled with junk. After all, your free `shinyapps.io` account has limited space.

The illustrative example is deployed online here. See also subsection below.

(Optional) Push to GitHub

One further thing we might want to do is to push the script files to `GitHub` as discussed in Chapter 4.

Try doing that yourself. I pushed my script files on a public `GitHub` repository that can be accessed here[5].

Progress illustrative example

Below is the entire code for the illustrative example that was transformed from a parameterized report into a shiny app.

You can try to replicate it by yourself and compare it with mine. Or copy paste it directly into a `app.R` script and run it. The choice is yours.

[4]Connecting to the server and deploying the app might take some time. Once the app has been successfully deployed, it will open automatically in your default web browser.

[5]Note that you need to be logged-in to your `GitHub` account to see the script files

```r
### calls r script files
source("functions.R")

### loads data
load("data/sample.Rdata")

### imports libraries
library(shiny)

r <- getOption("repos")
r["CRAN"] <-"https://cloud.r-project.org/"
options(repos=r)

# install.packages("pacman")

# pacman::p_load(tidyverse,readxl,haven,sjlabelled,kable,kableExtra)

#### shiny app starts here ###

# Define UI for application that draws a histogram
ui <- fluidPage(

    # Application title
    titlePanel("Illustrative example"),

    # Sidebar with a slider input for number of bins
    sidebarLayout(position = "left",

        sidebarPanel(

          # actor input
          selectInput("actor",
                      label="Choose an actor:",
                      c("Keanu Reeves",
                        "Alec Baldwin",
                        "Arnold Schwarzenegger",
                        "Timothee Chalamet",
                        "Anamaria Marinca"),
                      multiple = TRUE),
          # stereotype input
          selectInput("stereotype",
                        label="Stereotype dimension:",
                        c("Warmth women" = "wom_warm",
                                  "Competence women" = "wom_comp",
                                  "Warmth men" = "men_warm",
                                  "Competence men" = "men_comp"))

        ),

        # Show a plot of the generated distribution
        mainPanel(

          h3("Output displayed here",),
          # start tabset Panel
          tabsetPanel(

            # tab 1
            tabPanel("Actors",

                     tags$p(HTML(paste("A table is generated based on the actors chosen on the

                     DT::dataTableOutput("act") ),

            # tab 2
            tabPanel("Stereotypes",

                     tags$p(HTML(paste("A plote is generated based on the variable chosen on tl

                     plotOutput("st") )

          ) # close tabset Panel
        )
    )
)

# Define server logic required to draw a histogram
server <- function(input, output) {

  ####### -- imports and prepares data from here
```

```r
# reactive object
# data from Stanciu et al. 2017

tempdf <- reactive({

  choice=input$stereotype

  dfex %>%
    sjlabelled::remove_all_labels() %>%
    pivot_longer(contains("warm") | contains("comp")) %>%
    filter(name %in% choice)

})

# reactive object
# meta data movies
movietmp<- reactive({
  dfmv<-readxl::read_excel("mat/movies.xlsx",1) %>%
    filter(Actor %in% input$actor)

})

  #### -- generates output objects from here

# generate ggplot
plottmp<- reactive({

  ## ggplot code
  (input$plot_type == "ggplot2")

  ggplot(tempdf(), aes(x=factor(gen),y=value)) +
    labs(title=paste0("Evaluation based on ", input$stereotype),
         x="Gender",
         y=paste0("Stereotype of ", input$stereotype)) +
    geom_boxplot() +
    theme_light()
})

##### -- code for output from here
```

```
  # render plot for user
  output$st <- renderPlot({

    plottmp()
  })

  output$act <- DT::renderDataTable({

    movietmp()
  })
}

# Run the application
shinyApp(ui = ui, server = server)
```

Note however that you'd still need to create all the external scripts and `r` data files.

Advanced resources

If you really really like `shiny apps` and want to master them, then this book by Hadley Wickham contains everything one needs to know. Other online resources are available and offer varying levels of complexity.

6 Parting words

This short book introduces the basics to getting started with using *R* beyond data analysis.

The book promotes an integrated workflow where `r` has a pivotal role. With the help of `r` and the associated tools, such as `RStudio` and `GitHub`, we have seen how we can create automatized reports to reduce repetitive tasks (Chapter 3). We have also seen that we can create our own websites and publish online books (Chapter 4). Finally, we have also seen that we can write web applications to engage with our audience (Chapter 5).

Taken together, and in due time, one can create for oneself a work routine that from the start adheres to open science practicies and in so doing makes their own work reproducible, transparent, and publically available.

R is a powerful programming language that can help in transitioning from close-ended data analysis software such as SPSS to programming-based data analysis. One benefit of this is that through `r` one can discover a universe of new possibilities – for example, linking data analysis with communication strategies.

`r` is but one of the possible programming languages. The more one becomes accustomed with a programming-based logic in handling data, the easier it gets to integrate other programming languages into their workflow. For example, through their integrated development environment (IDE) of choice – in this case we've used `RStudio`.

`RStudio` facilitates the seamless integration of various programming languages (e.g., `r`, `python`, `Julia`) in enhanced documents (e.g., `RMarkdown`, `Quarto`) from which powerful and dynamic output files can be generated.

This short book was meant as a beginner's guide. Throughout the book advanced resources have been introduced.

And remember...

> Begin small but, aim high.

Advanced resources

If you still are not convinced of how powerful *R* can be, then what about this: You can create art using `r`. That is right! There are several packages available that can help you get in touch with your creative side while not ignoring your love for data!

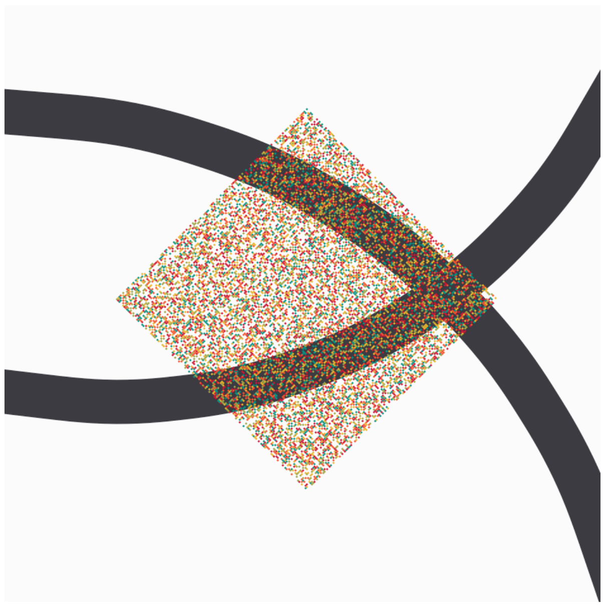


Figure 6.1: Example of `r` generated art using the package aRtsy.

`aRtsy` package is downloadable as explained here and attributed to Derks (2023).

Danielle Navarro provides a workshop for art from code using *R*: see here.

About the author

Dr. Adrian STANCIU is assistant professor in lifespan developmental psychology (focus on adult development and digitization) at the University of Luxembourg. He studies how through digital technologies the life quality and health of older migrants can be improved and sustained longterm. He is likewise researching on human values, mental health, and methodology of assessment, often in international collaborations and in an interdisciplinary approach.

Some of the images used throughout this short book were generated using Microsoft Copilot.

References

Conner, M., and Christopher J. Armitage. 1998. "Extending the Theory of Planned Behavior: A Review and Avenues for Further Research." *Journal of Applied Social Psychology* 28 (15): 1429–64. https://doi.org/10.1111/j.1559-1816.1998.tb01685.x.

Derks, Koen. 2023. *aRtsy: Generative Art with 'Ggplot2'*. https://CRAN.R-project.org/package=aRtsy.

Stanciu, A., C. J. Cohrs, K. Hanke, and A. Gavreliuc. 2017. "Within-Culture Variation in the Content of Stereotypes: Application and Development of the Stereotype Content Model in an Eastern European Culture." *The Journal of Social Psychology* 157 (5): 611–28. https://doi.org/10.1080/00224545.2016.1262812.

Witte, E. H., and A. Stanciu. 2023. "Error Theory of Mental Test Scores Without and with a Measurement Instrument." https://doi.org/10.31219/osf.io/9ap6m.

Witte, E. H., Stanciu A., and F. Zenker. 2022. "Predicted as Observed? How to Identify Empirically Adequate Theoretical Constructs." *Frontiers in Psychology* 13: 980261. https://doi.org/10.3389/fpsyg.2022.980261.