# Chapter 2

# Short Introduction to R

Gabriela Lunansky[1], Sacha Epskamp[1,2], & Adela-Maria Isvoranu[1]

1. University of Amsterdam, Department of Psychology
2. University of Amsterdam, Centre for Urban Mental Health

Do not share

## 2.1 Introduction

This chapter aims to provide the reader with the programming background and skills necessary to independently carry out the practical exercises and analyses in this textbook. To date, the network approach is almost uniquely implemented in R, making it an essential tool for carrying out psychological network analyses. Readers who are already familiar with R can skip this chapter and continue with Chapter 3 of this textbook.

## 2.2 The R environment

### Why use R?

Using R for psychometric analyses is no longer reserved for specialists or extremely enthusiastic methodologists. During the past decade, psychometric analyses in R have expansively increased, making it, currently, the norm for doing general psychometric analyses (Mair, 2018). There are good reasons for this: R is a powerful programming language for statistical analyses, data visualization, data mining, and general programming. Furthermore, R is freely available, open source, and is accompanied by a large and lively community. Many statistical analyses, such as the ones discussed in this book, are implemented in R packages, making the use of novel statistical methods easy.

Cite this chapter as:

Lunansky, G., Epskamp, S., & Isvoranu, A. M. (2022). Chapter 2. Short introduction to R. In Isvoranu, A. M., Epskamp, S., Waldorp, L. J., & Borsboom, D. (Eds.). *Network psychometrics with R: A guide for behavioral and social scientists*. Routledge, Taylor & Francis Group.

## Installing and setting up R and RStudio

We suggest installing two different programs for using R: the base program R itself, as well as an integrated development environment (IDE): RStudio. R is the base programming language, which can be operated through the terminal or command prompt. The popular IDE RStudio presents a clear and easy-to-use environment to work in, which includes in addition to the R console itself also a plain text editor for editing R code, several useful panes for displaying plots, help files, loaded objects, and more. While it is possible to carry out all analyses presented in this book without using RStudio (e.g., by directly working in the computer terminal), we suggest using RStudio and assume it is used throughout this book.

Please note that R and RStudio need to be installed separately from different websites, and that these programs have to be updated separately as well. The latest version of R can be installed from https://www.r-project.org/, and the latest version from RStudio can be installed from https://rstudio.com/products/rstudio/download/. It is advisable to always work with the latest version of both programs.

After installing both R and RStudio, you can open RStudio to get started.[1] It can then immediately be seen that the RStudio program consists of four main panes:

**Console** The console pane processes commands, meaning that you can enter and execute commands here. However, we suggest using the Source pane for executing (and saving) your commands. The console is similar to the terminal of the R base program and it will not allow saving your script.

**Source** The source pane is a plain text editor, meaning it shows plain text in the form of numbers, symbols, and letters. Here, you can write and edit your code, as well as save it in the form of a script. It is highly recommended to work from your script and send your commands to the console, instead of directly working from the console.

**Plots/Help** This pane has two functions. The first is to show graphical output, for example, figures such as scatterplots or barplots created from your code. Graphical output can be saved in several formats, such as `.pdf`, `.tiff`, or `.jpeg`. The second function of this pane is to show you documentation of functions (functions will be further discussed in Section 2.5). Any function in R is accompanied by documentation. Documentation gives information on what functions do, which arguments can be specified, and what the output will be. Many manuals also include example code on how to use the function. Documentation for every function can be found under the `help` tab or by using the question mark followed by the name of the function `?name`. For example, running the following command `?mean` gives the documentation for the function which calculates the mean of an object.

**Workspace** The workspace shows which objects are currently loaded in your environment. Objects can consist of, for example, loaded data sets or created objects

[1] Note: on some operating systems, an icon labeled "R" will also be created. It is important to note that this usually is not R, but rather an older IDE that comes with the default R installation. We recommend to use RStudio instead of this IDE.

Do not share

such as matrices and vectors. Please note that it is not recommended to save your workspace when closing RStudio, as in every future session the same objects will be reloaded and this may create issues in new scripts or overwrite functions. In addition, always loading your objects from your script is better practice for reproducibility.

It is recommended to have the console and source panes on top, since they are the most important, and plots and workspace panes below (see Figure 2.1). Pane layouts can be set with *Tools → Global Options → Pane Layout*. In *Tools → Global Options → Appearance* the theme of RStudio, including background colors and text colors, can be changed.

## 2.3 Basics of R programming

This section will discuss some of the basics of R programming, starting with how R can be used as a calculator, and ending with how to write the first forms of an R script that can be saved and loaded later.

### Using R as a calculator

To start programming in R, experiment with the console pane. For example, write the following command, followed by pressing `enter`:

```
1 + 1
```

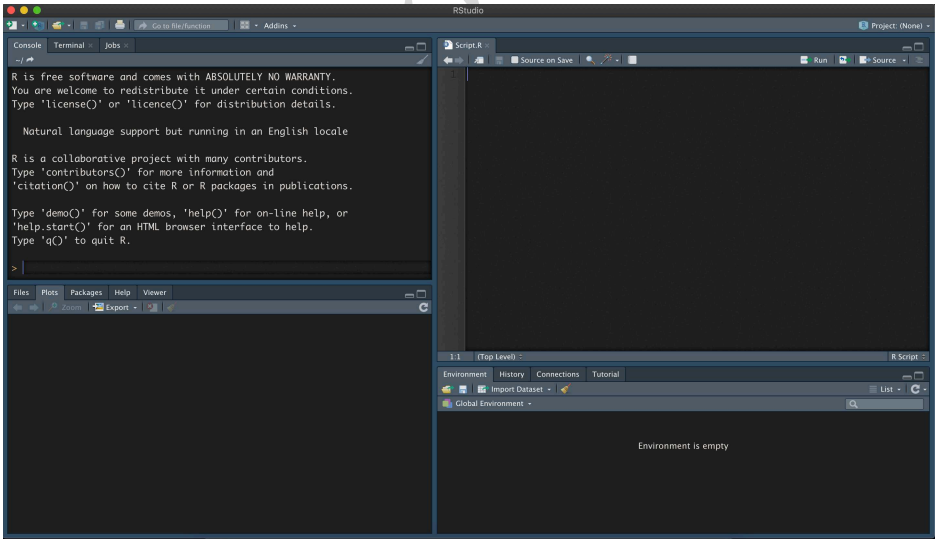



Figure 2.1. Example of RStudio with the four panes ordered in the recommended setup, clockwise from top left: the R Console pane, the Script pane, the Workspace pane, and the Plots/Help pane. Note that the color of RStudio is set here according to a theme, which can be set using *Tools → Global Options → Appearance*.

R will now return the number 2, telling you that the sum of 1 and 1 equals 2. The full output in the console is:

```
[1] 2
```

The `2` refers to the answer of our question ('what is 1 + 1?'). The `[1]` indicates that the first element of the answer is the number 2. This is specific to R: R will often return *vectors*, series of numbers, even if the vector is only one element long (i.e., a single number). So in full, this output tells us that a vector is returned, and that its first element equals 2. For now, you can ignore the `[1]` completely.

Next, try to enter:

```
1 +
```

followed by pressing *enter*. Now, something else happened: the console prompt > changed into +. This is because R is still expecting input. If you type another number and press `enter` the command will finish and output will be returned. Alternatively, if you press `escape` the command will be canceled, allowing you to enter new commands into R.

Of course, R can do more than just add numbers. Several other operators work as might be expected. For example, `*` can be used to multiply numbers, `/` can be used to divide numbers, `^` can be used to raise numbers to some power, and brackets `(` and `)` can be used to group commands and give precedence over evaluating some commands first. In addition, there are several *functions*, which will be explained in more detail later, that can be used for mathematical operations: `sqrt(...)` computes a square root ($\sqrt{\ldots}$), `exp(...)` computes the exponential function ($e^{\cdots}$), `log(...)` computes a logarithm ($\ln(\ldots)$), and so forth.

## Writing R code in a script

While the console is the pane that actually accepts R commands, as well as the pane in which most output is returned, it is highly recommended to never write code in the console pane directly. Instead, R code should be written in an R script such that the commands can be saved and used later again. In addition, scripts allow for more complicated sequences of R commands to be written. Often, you need to evaluate many commands sequentially, e.g., commands to read, transform, plot, or analyze data. Furthermore, saving code in scripts makes your code reproducible for other researchers.

To create an R script, select *File* → *New File* → *R script*. Immediately also save the script using *File* → *Save As*. Always use the `.R` extension to save R scripts. Now, R commands can be written in the script pane instead of the console pane. Sending commands from the source pane to the console is done by selecting the relevant code and pressing "Run", or alternatively, pressing `control+enter`/ `cmd+enter`. This sends the selected commands to the console, where they will be processed.

## Comments

Including comments in your script helps other researchers, or your future self, understand your code. Comments are added by using the hashtag # before writing code; any line of R

Do not share

code will no longer be evaluated from the moment # is encountered. These comments can then be used to explain what the purpose of the code is or why it is added. Alternatively, comments can be used to omit part of the code temporarily (this is termed 'commenting out' some code). Tutorial Box 2.1 gives some examples of how a comment can be used.

Any code past a # symbol will not be evaluated by R. For example the code:

```
1 + 1 # + 1
```

will return 2, not 3, as the code `1 + 1` is evaluated, and the last bit is not. As such, the # can be used to clarify code. For example, a clarification can be added after a line of code:

```
1 + 1 # This sums the numbers 1 and 1
```

or before a line of code:

```
# Sum the numbers 1 and 1:
1 + 1
```

We recommend to use comments very liberally in the code. For example, it is not uncommon for almost every line of R code in an analysis script to be accompanied by a comment.

Tutorial Box 2.1. Placing comments in the code to clarify code.

## Programming style and coding conventions

Carrying out statistical analyses in R implies writing code. This means not only having to learn the correct programming commands for conducting specific analyses, but also learning rules on what code should look like. The set of rules that determine the coding format are called coding conventions, which are related to general programming style. Just like there are guidelines for how to write research papers (e.g., American Psychological Association, 2020), many guidelines also exist for how to write clear code. While it is never mandatory to adhere to a specific coding convention, writing code according to coding conventions does greatly increase its readability and clarity, making it easier to share with other people and to maintain code over time.

Although there is no clear best style guide for programming in R, the "tidyverse style guide" by Hadley Wickham is often viewed as one of the most important ones.[2] It should be noted, however, that style guides are inherently opinionated, and therefore arguments can be made in favor and against every element included in any style guide. This book will not rely on strong rules on particular programming style. However, we would like to make readers aware that style guides exist, and encourage thinking about the format of code when doing analyses in R.

[2] https://style.tidyverse.org/

Do not sha

## 2.4 Basic R data structures

While it certainly can be nice and useful to write mathematical expressions in R, the output of such expressions has to be stored such that it can be used in later commands. In addition, when data are analyzed, they need to be stored inside R as well, such that they can be called later. Saving the output from commands or storing data in R can be done by using *objects*. This section discusses objects in more detail.

### Assigning objects

Objects can contain anything in R. For example, a single number can be stored in an object, but also more complicated data structures and books worth of text. The <- operator is used to store values into objects. For example, the following statement saves the number 1 in the object `a`:

```
a <- 1
```

Now, instead of writing 1, the object `a` can be used. For example:

```
1 + a
```

will again return the number 2. Alternatively, the = operator also functions to store values into objects. We do not recommend using =, however, because <- is unambiguous, explicating the left part becomes what is on the right part, while the = operator is ambiguous. Furthermore, = cannot be used within function calls to assign objects.[3]

R expressions that are not stored into an object are printed in the console. When entering a command where an expression is saved into an object, the result of the object is not printed in the console. To see the value of the object, you need to run the name of the object into the console. For example, the following statement evaluates 1 + 1 = 2 and stores the result into the object `b`, but does not print the result:

```
b <- 1 + 1
```

To get the result, we can ask R to print the result with `print(b)` or just by typing `b`. All currently loaded objects are listed in the workspace pane. When quitting R, you will be asked if you would like to save the objects in your workspace. As emphasized above, it is recommended to never do this.

### Naming objects

You can use any combination of letters, numbers, and some symbols (_ and .) to form object names, as long as the object does not start with a number. It is important to give your objects informed names, which describe the content of your object. In this way, it will be easier for other people to read and understand your code. Various conventions and standards exist regarding object naming. Object names in R are case sensitive: `object`, `Object`, and `OBJECT` are different objects. The tidyverse style guide recommends to only use lowercase letters, numbers, and underscores. Underscores should be used to separate

[3] In function calls, = assigns function arguments instead of objects.

Do not share

words, for example, `data_wave_1` instead of `datawaveone`. Another frequently used style, without using underscores or other operators, is *lowerCamelCase*. Here, the first letter of the first word is lowercase, and the following first letters of words are uppercase. For example: `sampleSize`, `rawData`, and `nMales`.

## Object modes

Objects in R can be of different types, called object modes. There are three main object modes for the standard output in R (a vector): numeric, character, and logical. Tutorial Box 2.2 gives examples of each of these object modes. Numeric objects store numbers, on which you can apply mathematical operations. Character objects, also called strings, store any form of text between tick marks (which can be double tick marks, `"`, or single tick marks, `'`). You cannot apply mathematical functions on character objects, since character strings are not numbers (even if they look like numbers). Logical objects indicate if something is true or false, and consist of the Boolean objects `TRUE` or `FALSE`. Commonly used shortcuts you may see in code are `T` and `F`. However, we strongly advise not to use these shortcuts. This is because `T` can be overwritten and stored as another object, for example `F <- 1`. In this case, `F` no longer refers to `FALSE` but to the number 1. This is problematic, as R sometimes interprets the number 1 as `TRUE` without warning. When a logical statement is preceded by an exclamation mark (`!`) the logical result is reversed (`TRUE` becomes `FALSE` and `FALSE` becomes `TRUE`). Logical modes are used in two ways: in functions, to assign logical modes to arguments, and in performing logical tests.

Sample

The following code creates a *numeric* object:

```
numericObject <- 1
```

This object can subsequently be used as a number:

```
numericObject + 1 # This will return the value 2
```

The following object is a string and not a number:

```
characterObject <- "1"
```

This object cannot be used as a number:

```
characterObject + 1 # This will result in an error.
```

Finally, we can also store a logical object:

```
logicalObject <- TRUE
```

Interestingly, this object *can* be used as a number, as `TRUE` also refers to *1* and `FALSE` to *0*. In addition, the logical values are also returned when asking R a logical test. For example:

```
1 == 0 # This will return the message FALSE
```

Tutorial Box 2.2. Examples of three object modes that can be used in R: numeric, character, and logical.

### Missing data

Missing data are encoded in R with `NA`, which means 'not available.' Missing data are handled differently for different functions. To find the right argument for handling missing data in the function you are using, review the documentation for that specific function using `?`.

### Vectors

A vector is an object that stores multiple values. These values can include numbers, but also characters such as letters, or missing data (encoded with `NA`). To assign a series of values to a vector, you can use the *combine* function `c(...)` by simply adding the series you want to combine into a vector. Every element of the series is separated by a comma. A second way to create a vector, is to use the colon `:` to create a series of successive numbers. Vectors can also be *indexed* using square brackets (`[` and `]`), which means selecting a certain cell or collection of cells from the vector. This can be used to check the values of these cells, to use them for analyses, or to change them. Tutorial Box 2.3 gives some examples of how vectors can be used.

## 2.5 Functions and packages

The introduction of vectors also introduced the concept of *functions*. A function is a small program: it takes input, does something, and gives output. For example, the *combine* function `c(...)` takes some values, combines them, and gives a vector consisting of those values as output. Some other examples of common functions are `mean(x)`, `sd(x)`, and `sum(x)` to compute the mean, standard deviation, and sum over elements of a vector called `x`. Functions always have the same form, namely, their name, and then the corresponding arguments within brackets: `name(argument, argument, argument, ...)`. Some arguments have default settings, meaning that the parameters of that argument are set. This makes the functions more generic to use, however, you can always change the parameters of every argument when using functions.

To know which arguments can be provided, and which default arguments apply, every function in R is accompanied by documentation. Reference cards also give a good overview of R functions for frequently used analyses, which can be found online. Furthermore, you can use online search engines and the R community to find the exact function names and arguments for the analyses you want to execute. You can also write your own functions using `function()`. However, this is part of advanced programming skills and will not be discussed throughout this book. Several freely available online resources on this topic exist.

Packages are extensions contributed to R containing extra functions. Developmental versions of R packages can often be found on Github, and stable versions of R packages are commonly stored on the Comprehensive R Archive Network (CRAN). Several packages are necessary for carrying out the analyses described in this book. All packages

Do not share

The combine function (`c`) can be used to create a vector. For example, the following code will create the vector $\begin{bmatrix}1 & 2 & 3 & 4 & 5\end{bmatrix}$:

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

Because this is an integer sequence, the same vector could also be formed using:

```
numericVector <- 1:5
```

A vector can also contain missing elements, which are encoded as `NA`:

```
vectorMissing <- c(1, 2, 3, NA, 5, 6, NA, 8, 9, 10)
```

Many functions in R tend to return errors when elements of a vector are missing. For example, the `mean` function can be used to compute the mean of a vector, but using it on a vector with missing elements will lead to an error:

```
mean(vectorMissing) # Results in an error
```

Instead, missing data must be handled in a way that is acceptable for the function used. For the example above, we could remove all missing elements using `na.omit`:

```
mean(na.omit(vectorMissing))
```

or an argument of the `mean` function can be used to do the same:

```
mean(vectorMissing, na.rm = TRUE)
```

Always refer to the help file of a function for information on how to handle missing data. Finally, vectors can be indexed using square brackets that follow the name of the vector. The square brackets can contain integers indicating the elements of the vector (starting with 1), or a logical vector of the same length as the original vector with `TRUE` indicating an element should be returned. For example, `vectorMissing[1:3]` selects only the first three elements of the vector and `vectorMissing[!is.na(vectorMissing)]` selects all elements that are not `NA`.

Tutorial Box 2.3. Examples of how to use vectors in R.

need to be installed once by using `install.packages()`.[4] For example, the following command installs the package *bootnet*, which we will use in the remainder of this book:

```
install.packages("bootnet")
```

This may take a long time, and only has to be run occasionally, to ensure the package is up to date. It is not recommended to leave a call to `install.packages(...)` in an R script (or if it is included, include a comment sign `#` before the call so it is not always evaluated). After installing a package, the package also needs to be loaded using the

[4] Sometimes R may struggle to install an R package and return an error. If R asks if you want to install a package from source, it is best to answer 'no' unless you have your system set up to install packages from source. Answering 'no' does not stop installation but rather continues with installing pre-compiled packages from CRAN, which is easier. Most R packages depend on other R packages, which should be installed automatically. Sometimes, however, this does not work. Whenever R returns an error that some package is not installed or cannot be loaded, try installing that package first. Sometimes then a new error may be returned pointing out a different package that needs to be installed.

function `library(...)`.[5] For example, the *bootnet* package can be loaded with:

```
library("bootnet")
```

This makes all exported functions from the *bootnet* package available to the user. Contrary to `install.packages(...)`, the `library(...)` command does have to be used at the beginning of every script that uses functions from a package.

## 2.6 Advanced object structures

We already discussed different modes of objects in R and vectors that combined several objects in Section 2.4. Typically, data in R are presented in more advanced structures than vectors. For example, networks can be encoded using two-dimensional matrices, and data are typically stored in a form of a matrix, in which columns can have different modes (data frames). This section will introduce these more advanced object structures.

### Matrices

An important function when doing network analyses is the `matrix()` function. Technically, a matrix is a vector with two dimensional attributes. Rows indicate horizontal lines of cells, while columns indicate vertical lines of cells. The first argument of a matrix is a vector (i.e., the data) to fill the matrix with, the second argument the number of rows, and the third argument the number of columns. Indexing in matrices can be done using square brackets, where the first value indicates the row number and the second value the column number, separated by a comma. Some examples of how to use matrices are shown in Tutorial Box 2.4, and documentation for the matrix function can be found with `?matrix`.

The `matrix` function can be used to create a matrix:

```
myMatrix <- matrix(1:9, nrow = 3, ncol = 3)
```

Note that the matrix is filled columnwise. To fill the matrix rowwise, we can instead use:

```
myMatrix <- matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
```

Next, we can index the matrix using square brackets. For example, `myMatrix[1,2]` selects the cell at row 1 and column 2, `myMatrix[1:2,1:2]` select the block with the first two rows and columns, and `myMatrix[1:2, ]` selects the first two rows and all columns.

Tutorial Box 2.4. Example code for creating and indexing matrices.

[5] The function `require()` can also be used to load an R package.

do not share

We can create a list with two objects, one vector consisting of character elements, and one matrix consisting of numeric elements, with the following code:

```
myList <- list(
  characterVector = c("a", "b", "c", "d"),
  numericMatrix = matrix(data = c(1, 2, 3, 4),
  nrow = 2, ncol = 2))
```

To confirm that the list consists of two objects, one character vector and one numeric matrix, we can use the structure function with `str(myList)`. To index the list, we can use double square brackets. For example, the following selects only the first element of the list:

```
myList[[1]]
```

We can do the same by using the name of the element:

```
myList$characterVector
```

Finally, we can also index elements of lists. For example, the following selects the first element of the first object of the list:

```
myList[[1]][1]
```

Tutorial Box 2.5. Examples of constructing and indexing lists.

Do not sha

## Lists

Lists can consist of any (combination of) data modes. Lists are like coat racks that can store any type and combination of object modes, even other lists! For example, numeric matrices and vectors consisting of characters can be combined into a list. The first element will be the matrix, while the second element will be the vector. Every element of a list can be named. To index from a list, the `$` operator or double square brackets (`[[` and `]]`) can be used. An example can be seen in Tutorial Box 2.5.

## Data frames

Data frames are the most used objects for storing data sets; most data sets, such as data sets imported from the statistical program SPSS, will be represented as data frames in R. Technically, a data frame is a list in which each element is a vector of the same size. This allows the data frame to also be represented (and indexed) as a matrix. Data frames can store different variables, such as numeric variables, indicating scores on a questionnaire, but also characters, such as `"male"` and `"female"`. Since they are technically a list, you can index data frames using the `$` operator. However, you can also index data frames in the same way matrices are indexed. Tutorial Box 2.6 shows how a data frame can be created.

We can create a data frame with two variables indicating a participant's score and gender, consisting of a numeric vector and a character vector:

```
myDataFrame <- data.frame(
  score = c(3, 6, 2, 4),
  gender = c("male", "female", "other", "female"))
```

To index the data frame, the `$` operator, single square brackets and double square brackets can be used. For example, all these commands will index the second column (gender):

```
myDataFrame$gender
myDataFrame[,"gender"]
myDataFrame[["gender"]]
myDataFrame[,2]
myDataFrame[[2]]
```

A popular R package for manipulating data frames is the *dplyr* package (Wickham et al., 2021). The *dplyr* package contains several powerful functions for manipulating data. For example, the `select` function can also be used to select a variable:

```
select(myDataFrame, gender)
```

and the `group_by` and `summarize` commands can be used to compute something for every level of one or more grouping variables:

```
summarize(group_by(myDataFrame, gender), mean = mean(score))
```

This code computes the mean for every level of `gender`. The *dplyr* package exports a handy *pipe* operator, `%>%`, which can be used to express, for example, `f(g(x))` as `x %>% g %>% f`, in which `x` is some R object and `f` and `g` are some R functions. This can be used to write chains of commands easier (reading from left to right instead of from inside to outside):

```
myDataFrame %>% group_by(gender) %>% summarize(mean = mean(score))
```

The pipe operator syntax is used in some of the examples of this book. Of note, recent versions of R also include the pipe operator as `|>`. The *dplyr* package is part of a collection or R packages that all work well together, termed the *tidyverse* (Wickham et al., 2019). Other popular tidyverse packages are *ggplot2* for data visualization, and *tidyr* for data reshaping.

Tutorial Box 2.6. Working with data frames.

## 2.7 Working with data in R

Now that you are familiar with the way R works, it is time to start looking at real data. To do this, data need to be loaded into R as an object, after which it can be used for analysis.

### Working directory

The working directory is the folder on your computer in which R is currently operating. This means that your data will be loaded from this folder, and any output created (e.g., new data and plots) will be saved to this folder. You can request your current working directory

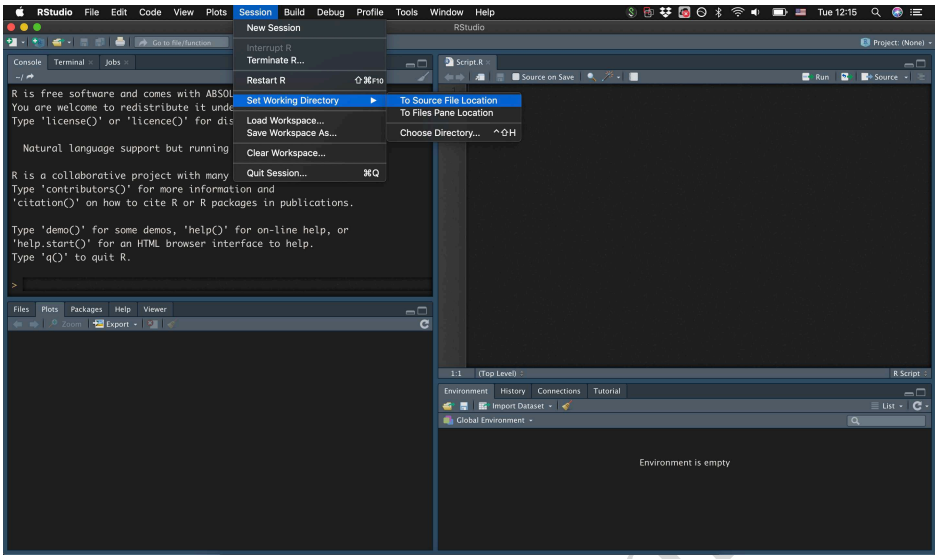



Figure 2.2. Setting up the working directory.

with the function `getwd()`. A common error that can occur is that a file is requested which is not located in the working directory. To solve this issue, you need to change your working directory or add the needed (data)file to your current working directory. To set your current working directory, you can use the function `setwd()`. Another way to set your working directory is by clicking *Session* → *Set Working Directory* → *Choose Directory*. This will allow you to set your working directory to a location of choice. Finally, as also shown in Figure 2.2, you can also set your working directory by clicking *Session* → *Set Working Directory* → *To Source File Location*. This sets your working directory to the same folder as where your current script is saved, which is convenient, since this allows you to easily load and save objects, (data)files, and scripts within the same folder.

## Importing data into R

After the working directory is correctly specified, there are several ways to load data sets into R. First, data sets can be loaded into R from different sources (e.g., CSV, SAS, SPSS, Microsoft Excel) by clicking *File* → *Import Dataset*. This option will in addition provide you the code to import data, which you can also use directly. Second, data sets can be loaded into R directly, by using a command to read your data. The most common format to read data into R is the Comma-Delimited (CSV) format. Please note that reading SPSS or Microsoft Excel data files into R requires installing and loading dedicated R packages. Tutorial Box 2.7 shows examples of code that can be used to read data into R.

Do not share

If data `mydata` is stored in a plain text file with the extension `.csv` (comma separated values), and located in the same directory as the script, then, after setting the working directory to the same location as the script with *Session → Set Working Directory → To Source File Location*, the data can usually be loaded into R and stored in an object called `data` with:

```
# Read a CSV data file into R
data <- read.csv("mydata.csv")
```

Possibly, some arguments need to be used to specify how exactly the data are stored. See for more information `?read.csv`.

If the data `mydata` are instead stored in an SPSS file with the extension `.sav`, the following commands can be used to read the data and store them in an object called `data`:

```
# Read an SPSS data file into R
install.packages("foreign")
library("foreign")
data <- read.spss("mydata.sav", to.data.frame=TRUE)
```

Both the function `read.csv` and `read.sav` return the data in a data frame

Tutorial Box 2.7. Importing data from plain text files and SPSS files.

Do not sh

### Correlation & covariance

After the data are loaded into R, they can be used for statistical analysis. For example, the data could be used to estimate a correlation matrix, which will play an important role in some of the analyses discussed in this book. The function `cor()` computes a correlation from a data frame (or any other type of numerical data) and the function `cov()` computes the covariance. The argument `method` selects the method for computing the correlation or covariance. The default is using Pearson correlations, (`method = "pearson"`), but other options include Spearman (`method = "spearman"`) and Kendall (`method = "kendall"`) correlations.

## 2.8 Conclusion

This chapter introduced the reader to basic concepts of the statistical programming language R. The discussed topics should be sufficient for most of the R analyses discussed in this book. However, it can be noted that R is a very extensive programming language, and the full extent of programming in R cannot readily be captured in a single chapter. For example, we did not discuss actual programming in R (such as using `if` statements and `for` loops), how to write functions, how to write efficient R code, or how to do more complicated statistical analyses. As R is a very popular open-source programming language, there is a wealth of freely available guides available online that teach these more advanced topics of programming in R for the interested reader.

## 2.9 Exercises

### Conceptual

2.1. List three advantages of using R over using an interface based software such as SPSS. Can you also think of a disadvantage?

2.2. What are the three main object modes for a vector in R? Provide a short R-code example for every object mode.

2.3. The following code gives an error:

```
x <- "1"
1 + x
```

Explain why this error occurs. Can you transform the object `x` such that `1 + x` does not give an error?

2.4. The following code gives an error:

```
x <- c(1,10,NA,5,2,9)
sd(x)
```

Look at the help file of `sd` and find an argument that solves this error.

2.5. Given the following three vectors:

```
A <- rnorm(100)
B <- A + rnorm(100)
C <- B + rnorm(100)
```

Find out if the correlation between `A` and `C` is significant? What about the partial correlation between `A` and `C`, controlling for `B`?

### True or false

2.6. R is a freely available, open source program.

2.7. The source pane in R automatically processes commands, but does not allow saving your script.

2.8. The <- and = operators can be used interchangeably.

2.9. Using the shortcuts of logical objects (i.e., `T` or `F`) is recommended, as they are shorter and cannot be overwritten.

2.10. Packages need to be both installed and loaded every time you open R.

### Practical

For practical exercises in R, please navigate to the appropriate folder of this chapter, available on the online *Companion Website*.

Do not share

## References

American Psychological Association. (2020). *Publication manual of the American Psychological Association (7th ed.).* APA.

Mair, P. (2018). *Modern psychometrics with R*. Springer.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., ... Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, *4*(43), 1686.

Wickham, H., François, R., Henry, L., & Müller, K. (2021). *dplyr: A grammar of data manipulation* [R package version 1.0.7]. https://CRAN.R-project.org/package=dplyr

Do not share