R for Statistics

Reader

MAT27803

February 24, 2026

Preface

For the course "R for Statistics" (MAT27803) it is assumed that you have acquired knowledge about statistics. The minimal required level is "Advanced Statistics" (AS, MAT20306), "Advanced Statistics for Nutritionists" (ASN, MAT24306), or the course "Data Analysis for Biosystems Engineering" (DABE, MAT26806). If you do not have sufficient pre-knowledge, or your statistical courses were so long ago that memory traces have evaporated, you are strongly advised to follow a statistics course first. AS and ASN are presented in English.

The aim of "R for Statistics" is to provide an introduction to R, a computer language and environment for statistics and graphics. The course will focus on getting familiar with the R environment by means of RStudio, to use R for manipulation and exploration of data and to perform all statistical analyses learned in the basic and advanced statistics courses, i.e. simple linear and multiple linear regression models, analysis of variance for Completely Randomized Designs (CRD), Randomized Complete Block Designs (RCBD) and factorial designs, ANalysis of COVAriance (ANCOVA), non-parametric methods and analysis of categorical data. The correct interpretation and explanation of the results of preformed statistical analyses is expected based on the assumed prior knowledge. Basic programming, visualization, as well as reproducibility of results in R will also be part of this course.

Students with an interest in statistics may benefit from the courses "Data Analysis for Plant and Animal Breeding" (ABG30806), "Statistics for Data Scientists" (MAT32806), "Data Science for Plant Breeding and Genetics" (MAT33306) and/or "Bayesian Data Analysis" (MAT34806) as a follow-up. All these courses are presented in English and build upon AS, or ASN. In ABG30806 for example logistic and log linear models (for binary and count data), variance components models (for dependent data, both balanced and unbalanced), general inference techniques (maximum likelihood estimation, Wald test, likelihood ratio test), posterior Bayesian inference using Markov Chain Monte Carlo, and applications in genetics and epidemiology are discussed. Apart from these courses students are also welcome at Biometris for a MSc-thesis Mathematical and Statistical Methods (MAT80418, MAT80439).

The aforementioned courses are organized by or in cooperation with Biometris. Biometris is part of Wageningen University & Research and is responsible for a number of short courses for PhD students and participates in many other courses as well. Biometris is a partner in the M.Sc. course in Applied Statistics at Leiden University. Staff members from Biometris are actively involved in research, mainly in statistical genetics, systems biology and ecology, food health and safety, omics, and big data. We have, however, an interest in other topics as well. If you have an interest in statistics, or work on your master thesis will involve use of advanced statistics, you might consider doing your thesis in cooperation with Biometris. In that case please contact a member of staff of Biometris about this.

Contents

Course Information

First of all welcome to MAT27803. In the following sections you can read information about your participation in this course.

Educational aids

- Book: *An Introduction to Statistical Methods & Data Analysis* by R. Lyman Ott and Michael Longnecker, 7^{th} edition, CENGAGE Learning (available from Studystore). The 6^{th} edition , Brooks/Cole, may also be used.
- MAT27803 R for Statistics Reader
- Presentations (available as handouts via Brightspace, see below)

Brightspace site of MAT27803

On Brightspace, you will find, among other things, answers to exercises, handouts of the presentations. The easiest way to access this site is usually from your personal myWURtoday page. You need to have an account and to be registered for the course. The URL for Brightspace is https://brightspace.wur.nl

Attendance during MAT27803

During the course MAT27803 "R for Statistics" attendance is **compulsory** on Monday, Tuesday, Wednesday and Thursday. This is the case for any practical at Wageningen University & Research. On Friday attendance is voluntary, but recommended in case you have questions about the assignment or about previous computer lab days. Your attendance will be registered during the course, and you require a pass for attendance. In case of a fail for attendance you will receive an partially completed for the course, meaning you have not passed the course yet.

You are allowed to miss one computer lab day without any explanation. If you really have a sound and valid reason, you will be allowed to miss a second computer lab day. In case you know this in advance contact one or both course coordinators (see section Coordinators below), and explain why you will miss another computer lab. The lecturers will decide, whether the reason is good enough to miss a second computer lab day.

In case of three missed computer labs, the rules of baseball will be applied. Meaning, three strikes and you are out. This will cause you to immediately **fail** MAT27803 R for Statistics.

End Grade for MAT27803

The end grade for MAT27803 will be calculated on three assignments you **have** to hand in during the course.

There will be two end-of-the-week assignments (week 1 and 2), which will make up one-sixth each of your end grade. At the end of the third week of the course on Friday you will be given the end assignment (a.k.a. the exam) for MAT27803 "R for Statistics". You will have a full week to work on this assignment and it will make up two-thirds of your end grade for the course.

All assignments need to be submitted before or on the set deadline.

Failing to submit the end-of-the-week assignments on time will have consequences for your end grade. Submitting one day later will mean the grade of the assignment will only count for 80%, two days later 60% and so on. Submission after Friday morning will cause failing the course!

Failing to submit the end assignment on time will cause you to immediately fail the course.

The end grade will be a weighted average of the assignments, with the weights as defined in the above text. The validity of your end grade depends on whether you received a pass on attendance.

In case you have handed in (at least) one end-of-the-week assignment, which got graded above 5.5, you will get a partially completed as end grade. This still means you did not pass.

Coordinators

Name	Building/Floor	E-mail
dr. S. (Sara) Bahrami	Radix West (107) / 4^{th} floor	Sara.Bahrami@wur.nl
dr. M. (Maikel) P.H. Verouden	Radix West (107) / 4^{th} floor	Maikel.Verouden@wur.nl

About the Course Reader

In this reader for the course "R for Statistics" no prompts are added to R source code, and text output is commented out by default by `#>`. Package names are given in bold text, e.g. **stats** package. Function names are shown in italic (e.g., *paste()*), and object names are displayed in bold italic text (e.g. ***soapData***). Inline code, function arguments, and file names are formatted and displayed in a typewriter font (e.g., for inline code with function arguments: `mean(1:10, trim = 0.1)`, and a file name: `figure/foo.pdf`).

Whenever the term O&L is used to reference to theory, it means that this material can be found in the book of Ott & Longnecker as described in the section Educational Aids of the Course Information.

Part I

Week 1

Lesson 1

Getting Started

1.1 What is R?

Following the definition given on the official R Project website, R [1] is a language and environment for statistical computing and graphics. It is a GNU project (https://www.gnu.org/) similar to the S language and environment, which was developed at the now defunct Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues.

Therefore, to understand what R is, first an explanation has to be given what S is. S was initiated in 1976 as an internal statistical analysis environment – originally implemented as FORTRAN libraries. Early versions of the language did not contain functions for statistical modeling. In 1980 the first version of S was distributed outside Bell Laboratories and in 1981 source versions were made available. By 1984 the source code for S became licensed through AT&T Software Sales for education and commercial purposes. The system was rewritten in C around 1988 and began to resemble the system as in use today (this was Version 3 of the language). Version 4 of the S language was released in 1998 and is the version which is still in use today. The book *Programming with Data* by John Chambers [2] (also known as the green book) documents this version of the language. The fundamentals of the S language itself has not changed dramatically since 1998. Nowadays the company TIBCO Software Inc. (http://www.tibco.com/) sells its implementation of the S language under the product name S-PLUS (officially "TIBCO Spotfire S+"). This implementation has built a number of fancy features (Graphical User Interfaces, mostly) on top of it – hence the "PLUS".

R can be considered as a different implementation of the S programming language combined with lexical scoping. However, some important differences exist. Much code written for S runs unaltered under R. The development of R was started by Robert Gentleman and Ross Ihaka at the Department of Statistics of the University of Auckland, New Zealand around 1991. The name is partly based on the (first) names of the first two R authors (Robert Gentleman and Ross Ihaka), and partly a play on the name of the Bell Labs S programming language. In 1993 R was first publicly announced and since 1995 it is being developed under the GNU General Public License, making R free software (https://www.gnu.org/philosophy/free-sw.en.html). The year 2000 marks the release of R considered stable for public use and, therefore, labeled as version 1.0.0. Currently the development of R is done by the *R Core Team*, of which a.o. John Chambers, Robert Gentleman, and Ross Ihaka are members. The latest release is **R version 4.5.2 (2025-10-31)**, which goes by the name **[Not] Part in a Rumble** (all of the release names are references to Peanuts strips/films with the famous characters Charlie Brown, Snoopy and friends).

R provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity. The R system is conceptually divided into two parts:

1. The "base" R system.
2. Everything else.

R functionality is divided into a number of *packages*:

- The "base" R system contains, among other things, the **base** package which is required to run R and contains the most fundamental functions.

- The other packages contained in the "base" system include **utils**, **stats**, **datasets**, **graphics**, **grDevices**, **grid**, **methods**, **tools**, **parallel**, **compiler**, **splines**, **tcltk**, and **stats4**. Packages from the "base" system are only updated with a new release (major or minor) of R.
- There are also "Recommended" packages: **boot**, **class**, **cluster**, **codetools**, **foreign**, **KernSmooth**, **lattice**, **mgcv**, **nlme**, **rpart**, **survival**, **MASS**, **spatial**, **nnet**, and **Matrix**. These packages are not essential, but are called upon so frequently by other packages that they are highly recommended being installed. The difference between "base" system and "Recommended" packages is that "Recommended" packages can receive updates in between a new release (major or minor) of R.

And there are many other packages available:

- There are, nowadays, more than 21130 contributed packages on the Comprehensive R Archive Network (CRAN Contributed Packages), that have been developed by users and programmers around the world.
- There are also many packages associated with the Bioconductor project (https://www.bioconductor.org/).
- People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion.

The main features of R are:

- Syntax is very similar to S, making it easy for S-PLUS users to switch over.
- Semantics are superficially similar to S, but in reality are quite different.
- Runs on almost any standard computing platform/OS (even on the PlayStation 3). R compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and macOS.
- Frequent releases (annual + bug fix releases) and active development.
- Quite lean, as far as software goes; functionality is divided into modular packages.
- Graphics capabilities are very sophisticated and better than most statistical packages. The ease with which well-designed publication-quality plots can be produced, including mathematical symbols and formulae where needed, while great care has been taken over the defaults for the minor design choices in graphics (where the user retains full control).
- Useful for interactive work, but contains a powerful programming language for developing new tools. Therefore, providing a useful environment for both users as well as programmers.
- Very active and vibrant user community; R-help and R-devel mailing lists and Stack Overflow provide great examples of this active community.
- Availability of interfaces to other languages such as Python, Julia, Stan, etc.
- It's free software, which grants you the freedom, as specified by the Free Software Foundation (https://www.fsf.org/), to:
  - Run the program, for any purpose (freedom 0).
  - Study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
  - Redistribute copies so you can help your neighbor (freedom 2).
  - Improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

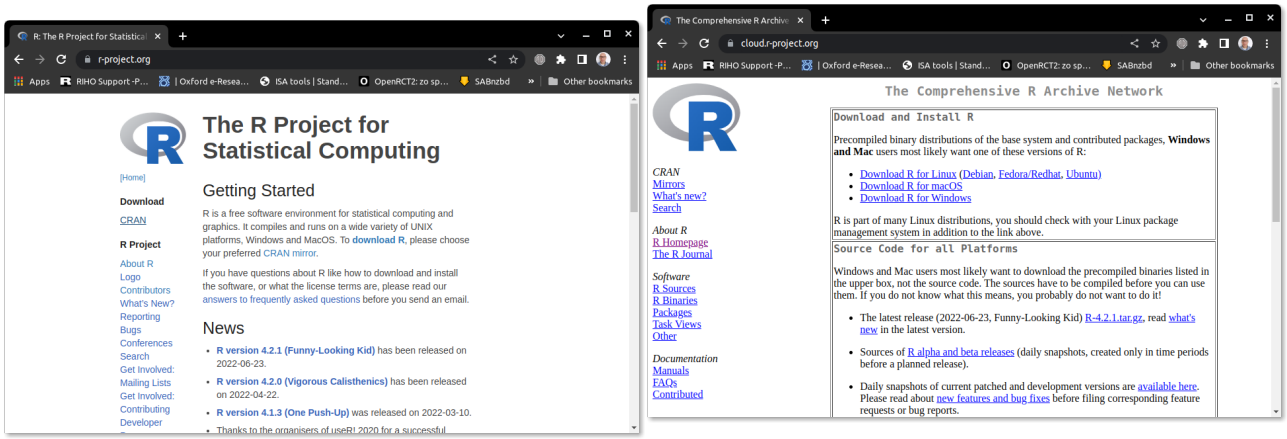
Some drawbacks of R are:

- Essentially based on approximately 40 year old technology.
- Little built-in support for dynamic or 3-D graphics (but things have improved greatly since the "old days").
- Functionality is based on consumer demand and user contributions. If no one feels like implementing your favorite method, then it's your job (Or you need to pay someone to do it)!
- Objects must generally be stored in physical memory; but there have been advancements to deal with this too (especially with respect to Big Data).
- Not ideal for all possible situations. However, this is a drawback of all software packages. There is no one size fits all!

1.2 Downloading and installation

For this course all required software (R as well as RStudio) has to be installed on your personal laptop computers. To install R on your home desktop or personal laptop the best starting point is the main website of

the R Project (https://www.r-project.org/). Figure 1.1a shows a screenshot of this main website.



(a) Screenshot of the R Project main website.

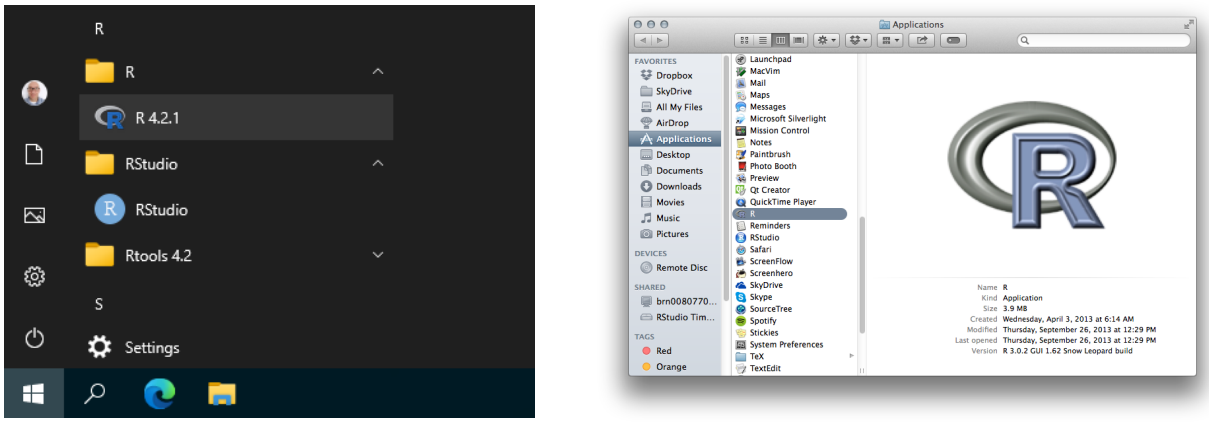
(b) Screenshot of the Comprehensive R Archive Network (CRAN) website.

Figure 1.1: Screenshots of the most important websites for downloading and installation of R.

In the left column menu of the main site under **Download**, as shown in Figure 1.1a, you will find a link to the Comprehensive R Archive Network (commonly referred to as CRAN), which will let you select the nearest mirror of CRAN. Figure 1.1b shows a screenshot of the CRAN website. Here you can download the latest version of R and find extensive documentation on how to install R on your machine (see section **Documentation** in the left side menu of the CRAN-website).

1.3 Starting R

How to start R depends on the operating system of the machine you are going to use. Users with the Windows or Mac operating system can use the default Graphical User Interface (GUI), which is installed by default with R. Figure 1.2a and Figure 1.2b display how to start the default R GUI, respectively, under the Windows and Mac operating systems.



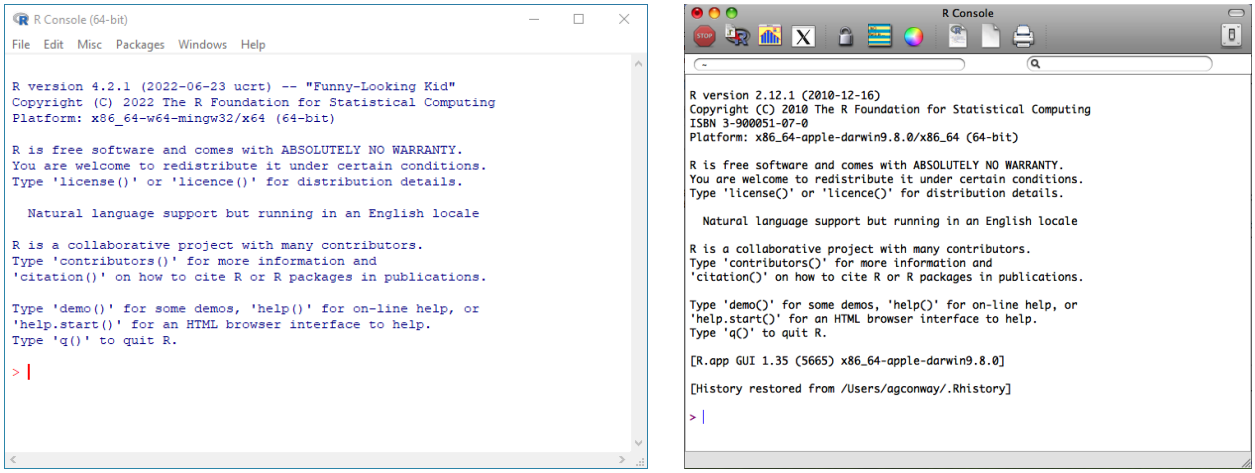
(a) Starting the default R GUI on Windows 10.

(b) Starting the default R GUI on macOS.

Figure 1.2: Screenshots of starting the default R GUI under Windows and macOS.

After invoking the R GUI under Windows or macOS a new program window is started as shown in Figure 1.3a and Figure 1.3b for, respectively, Windows and macOS.

For computers running Linux as operating systems there is not by default a GUI available. R can be started in a terminal session by typing `R` at the terminal prompt, as shown in Figure 1.4. In the same terminal a R session will be started as shown.



(a) Default R GUI on Windows 10.

(b) Default R GUI for macOS.

Figure 1.3: The default R GUI under Windows 10 and macOS.

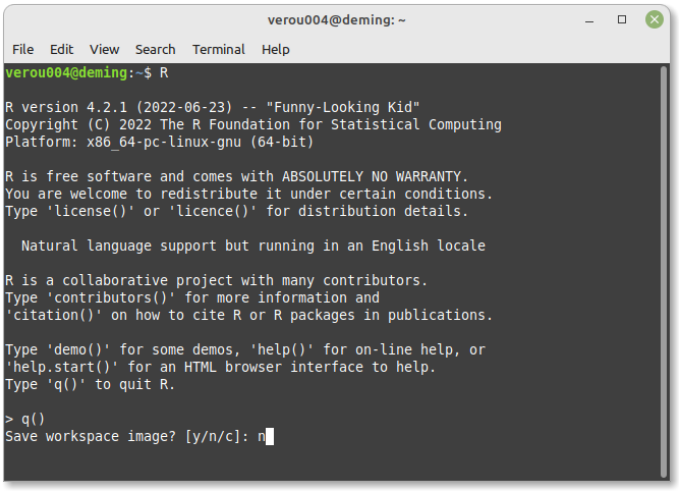


Figure 1.4: Starting R on computers with Linux as operating system.

1.3.1 R console Prompt

Although the environment, in which a R session is started, looks different for the various operating systems as displayed in Figure 1.3 and Figure 1.4. They all have one thing in common. Each R session starts at the console prompt, shown as **>**. The console prompt of a R session invites you to enter commands to be executed. These commands can be to calculate or transform some data or to display analysis results graphically.

1.3.2 RStudio

In this course a generic GUI will be used provided by RStudio [3], instead of using the default R GUI for the Windows and Mac operating system, or the terminal for computers running on Linux. The advantage of this GUI is, that it provides an integrated development environment consisting of 4 panels as shown in Figure 1.5:

1. Code Editor, with code highlighting to avoid typographical mistakes.
2. R Console.
3. History and Environment, *etc.* (in separate tabs).
4. Plots and Files, *etc.* (again all in separate tabs, for displaying graphs, easy file management, and browsing help and packages).

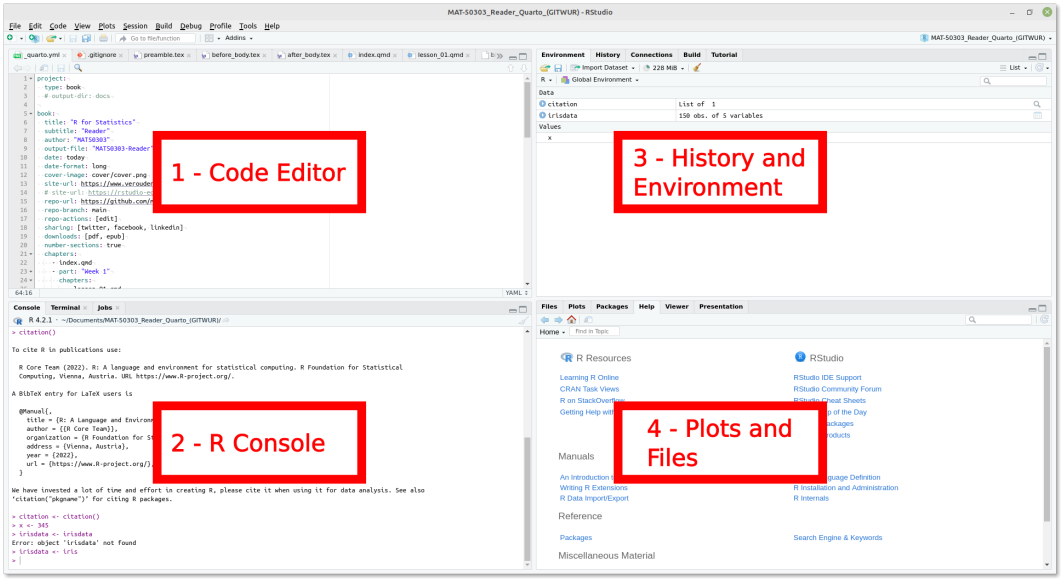


Figure 1.5: RStudio GUI: an integrated development environment.

RStudio can be downloaded and installed for **FREE** and the look and feel of the GUI is the same across all the various operating systems (Windows, macOS and Linux).

> **Note: RStudio Installation**
>
> In case you want to use RStudio as your default GUI on your home desktop or personal laptop, you need to install it **after** installing R.

To start RStudio you go to the start menu as displayed in Figure 1.2a and Figure 1.6 for, respectively, the Windows and Linux operating systems. For a Mac operating system RStudio can be started from **Applications** in a Finder window, as shown in Figure 1.2b (two lines below the default R GUI in this figure).

1.4 Using R interactively

When you start using RStudio (our GUI of choice for working with R), it issues a console prompt (Panel 2 of Figure 1.5) as described in Section 1.3.1. At the console prompt R can be used interactively, meaning that output of a command is printed in the console. You can for example use R as a simple calculator for subtraction of two values by typing the following at the console prompt and executing it by pressing **Enter** at the end of the line:

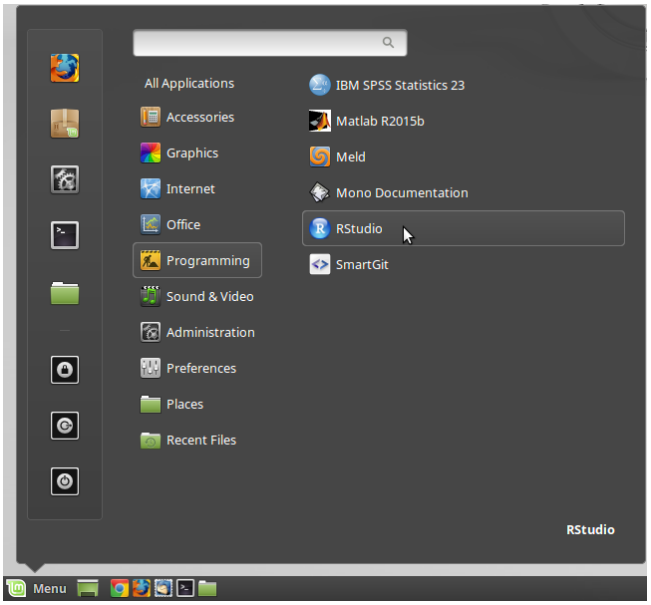


Figure 1.6: Start RStudio in Linux Mint 17.3 with a cinnamon desktop.

```
8 - 2
```

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

By pressing enter you tell R to evaluate your command and the output gets printed in the console (as shown above by the commented out part starting with `#>`). The `[1]` in the output tells which component in the output you are looking at. This is in our example not very useful, because the expected result is just one value.

1.4.1 Basic arithmetic operators

Apart from subtraction as used in the previous example, all basic arithmetic operators are available in R. Table 1.1 shows the symbols used to represent the basic arithmetic operations.

Table 1.1: Arithmetic Operators.

Operation	Symbol with example	Result
Addition	`9 + 2`	`#> [1] 11`
Subtraction	`9 - 2`	`#> [1] 7`
Multiplication	`9 * 2`	`#> [1] 18`
Division	`9 / 2`	`#> [1] 4.5`
Exponentiation	`9^2` or `9**2`	`#> [1] 81`
Integer division	`9 %/% 2`	`#> [1] 4`
Modulo / Remainder	`9 %% 2`	`#> [1] 1`

If a command is composed of several arithmetic operators, they are evaluated in the usual order of precedence, that is, first the exponentiation (power) symbol, followed by division, then multiplication, and finally addition and subtraction. Parentheses can also be added to control precedence if required or for readability of your code. For example in the following command the parentheses are used for readability:

```
3^2 + (6 / 3) + 2
```

```
#> [1] 13
```

While in this command the parentheses control precedence:

```
(3^2 + 6) / (3 + 2)
```

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

1.4.2 Functions

You will need to use functions in R to be able to do more than just basic arithmetic calculations, and exploit the full potential of R. In R functions are sets of commands with a given name and together perform a specific task, which produces some kind of output. In most cases a function also requires some input, e.g. data, which is supplied as function arguments.

In R there are many built-in functions, that perform a broad range of tasks from simple functions like rounding numbers to a number of significant digits, to importing files and performing complex statistical analysis. Throughout this course you will be introduced and will make use of these. Apart from using built-in functions, R can also be extended with new functions, which are available in packages or by creating your own functions. Both will be covered further on in this reader. Whenever you use a function, you will begin with typing the function name followed by a pair of parentheses. Any input required by the function, the so-called function arguments, is placed between the parentheses.

An example of a function, that does not require arguments, is *date()*. This function returns a character string of the current system date and time:

```
date()
```

```
#> [1] "Tue Feb 24 09:23:51 2026"
```

The function *round()* is an example of a function with arguments:

```
round(x = pi, digits = 2)
```

```
#> [1] 3.14
```

The given example rounds the mathematical value of π to two decimal places. Table 1.2 provides a few useful mathematical functions. Of course there are many, many more!

Table 1.2: A few useful mathematical functions.

Purpose	Function
Exponential	*exp()*
Natural logarithm	*log()*
Log base 10	*log10()*
Square root	*sqrt()*
Round	*round()*
Absolute value	*abs()*
Factorial	*factorial()*

1.4.3 Assignment operator and result printing

Now having introduced you to basic arithmetic operations and functions, you can already perform quite some calculations. Does this mean you have to type out commands in full all the time? Well of course not, R has a system that enables you to store results from calculations and graphs. Say you want to store the result of the subtraction given in Section 1.4 in an object named ***difference***. In R you would execute the following command:

```
difference <- 8 - 2
```

You could read this from right to left as: subtract 2 from 8 and assign this to an object named ***difference***. The arrow <- (constructed from a less than sign < and a dash -) is a so-called leftwards assignment operator, which stores the calculated difference (6) in an object named ***difference***. You see that, when assigning a command to an object, no output is generated in the console. To print the output to the console you can simply retype the objects name and execute it:

```
difference
```

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

This way of printing the value of an object in the console is called auto-printing. It can also be achieved by an explicit *print()* command:

```
print(difference)
```

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

Putting the full command between parentheses like this:

```
(difference <- 8 - 2)
```

assigns the result to ***difference*** and prints the result in the console in one go.

Objects, created with the assignment operator, are stored in your workspace environment. In RStudio objects can be seen on the tab named Environment in Window 3 as shown in Figure 1.5. The stored object names (like ***difference***) can also be printed in the console with the *ls()* or *objects()* function.

1.4.4 Incomplete commands

When an incomplete command is entered at the console prompt, the user gets warned in the R console by a + sign on the next line and a blinking cursor. You need to complete the command for proper evaluation. Try this yourself by omitting the 2 in the first subtraction example in Section 1.4. Complete the command by adding the omitted 2 and execute by pressing enter.

> Tip: Incomplete commands
>
> Incomplete commands often occur when nesting functions and not providing enough closing parentheses.
>
> When facing an incomplete command at the prompt as indicated on a new line in a R console with +, always first check the number of closing parentheses.
>
> A very useful feature in RStudio is using "Rainbow parentheses". You turn on this feature in the RStudio main menu under Tools > Global Options.... Select Code in the left column and the tab called Display. Place a check mark in the box in front of "Rainbow parentheses" to turn on the feature.

1.4.5 Finding answers and getting help

For sake of the learning process it is important to try to answer your own questions yourself first. The best way to find the answers to your questions is to google them. Google is your friend, when it comes to programming in general and in our case specifically to R. Almost every problem you encounter, when using and programming in R, has already been solved by somebody. So browse R user fora and search Stack Overflow for your answer. When you come across a question, you yourself have already solved please be kind and post a solution.

To find help about for example the *rnorm()* function (drawing random numbers from a normal distribution) in R you can execute at the console prompt:

```
help(rnorm)
```

or equivalently type and execute at the console prompt: `?rnorm`. This command will open the help browser to the specific function in Panel 4 as shown in Figure 1.5. In case you do not want to read the help page on a certain function, but just want to see what arguments the function, e.g. *rnorm()*, can take you execute:

```
args(rnorm)
```

```
#> function (n, mean = 0, sd = 1)
#> NULL
```

When you do know the name of the function and you want to see how it is implemented in R, you can see the code of the function, e.g. *colSums()*, by executing it without parentheses:

```
colSums
```

If you do not know exactly which function to use, you can search for all help files containing some word, for example "plot":

```
help.search("plot")
```

> ⚠ Warning: *help.search()*
>
> The *help.search()* function can produce a very long list of help files, in the help browser in Panel 4 as shown in Figure 1.5, containing the word you searched for.

1.5 Quitting R

When you start R, as described in Section 1.3, a few lines get printed in the console as can be seen in Figure 1.3 and Figure 1.4. One of these lines tells you the command to quit R. You can quit R by executing in the console:

```
q()
# or equivalently
quit()
```

Before R terminates, it first asks you whether you would like to save a workspace image containing all the objects you currently have in your workspace environment. When you answer this question with y(es) a `.RData`-file will be created in the directory you are currently working in.

1.6 Basics in R

This first chapter is meant as a first introduction and, therefore, written out in full. Reading about R is not the same as learning it, which in my humble opinion can only be learned hands on by working with it. From here on text will be kept much shorter and the focus will be on working with R.

1.6.1 Exercise SWIRL: Basic Building Blocks

he first exercise involves using the **swirl** package. **SWIRL** is an acronym for **S**tatistics **W**ith **I**nteractive **R** **L**earning. It provides an environment to learn R within R. To get started you first need to install the **swirl** package on the computer you are using. Execute at the prompt in the console the following command to install **swirl**:

```
install.packages("swirl")
```

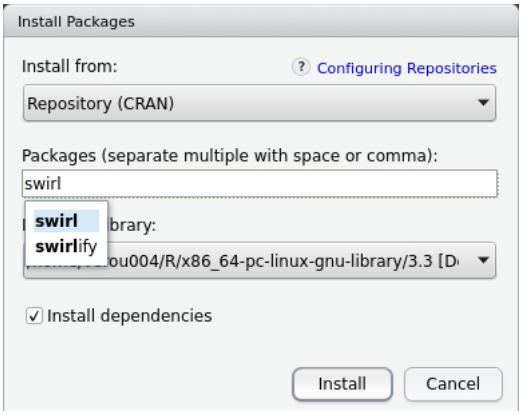


Figure 1.7: RStudio GUI for installing a package.

Alternatively packages can be installed in RStudio by using install on the packages tab, which is located in Panel 4 as shown in Figure 1.5. Selecting install will open a small window, as shown in Figure 1.7, where you can enter the name of the package you wish to install. Make sure you mark the tick box to indicate that all dependencies get installed as well. Finally, click the install button to install the desired package.

To be able to use the installed **swirl** package it needs to be loaded in R. Execute the following command to load the **swirl** package:

```
library(swirl)
```

```
#>
#> | Hi! I see that you have some variables saved in your workspace. To keep
#> | things running smoothly, I recommend you clean up before starting swirl.
#>
#> | Type ls() to see a list of the variables in your workspace. Then, type
#> | rm(list=ls()) to clear your workspace.
#>
#> | Type swirl() when you are ready to begin.
```

The package gets loaded and R gives you some valuable feedback on how to start up swirl as well as how to clean up your working environment.

When ready to start up swirl, execute at the prompt:

```
swirl()
```

The first time starting swirl will give a lot of output in the console. The most important things to do and keep in mind are:

- Providing swirl with a name, for example your given name. **Remember the name you provide in swirl, you need it each time you want to continue or restart a swirl lesson.**
- You can exit swirl and return to the R prompt **>** at any time by pressing the **Esc** key. If you are already at the prompt, type *bye()* to exit and save your progress. When you exit properly, you'll see a short message letting you know you've done so.
- When you are at the R prompt **>** during a swirl lesson:
  - Typing *skip()* allows you to skip the current question.
  - Typing *play()* lets you experiment with R on your own; swirl will ignore what you do...
    - UNTIL you type *nxt()* which will regain swirl's attention.
  - Typing *bye()* causes swirl to exit. Your progress will be saved.
  - Typing *main()* returns you to swirl's main menu.
  - Typing *info()* displays these options again.
- Install the "R Programming: The basics of programming in R" course, when the question arises which course to install.
- Select "R Programming" when you have to choose a course.

Having followed the steps given above you are now ready to start SWIRL Lesson 1, named **Basic Building Blocks**, by selecting it from the menu in the console.

1.6.2 Exercise: Navigating directories in R

In R it is important to know in which directory you are working, because everything happens relative to, what is called, the current working directory. In the header of the console, panel 2 of Figure 1.5, you can always immediately see what your current working directory is. Figure 1.8 shows a snapshot of the header of the console with the current working directory printed in the header.

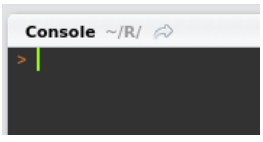


Figure 1.8: RStudio Console: Current working directory displayed in header.

There is also a function you can execute in the console to find out what your current working directory is:

```
getwd()
```

Try this command yourself and write down what your current working directory is. If you use R for the first time your current working directory will most likely be:

```
C:/Users/<YOUR_USERNAME>/Documents # On a Windows computer
/Users/<YOUR_USERNAME>            # On a macOS computer
/home/<YOUR_USERNAME>             # On a Linux computer
```

, where `<YOUR_USERNAME>` will represent your own username on that specific computer. You see that within R directories are separated by `/`, which indicates a Unix/Linux style for representing directories.

There are many ways to navigate directories and files. You can use the file manager of your operating system, in Windows this will be for example be File Explorer. This requires you to open this program first, if you do not have it open already. RStudio, versatile as it is, comes with a built-in file manager. Click on the **Files** tab in the lower right corner of RStudio, panel 4 in Figure 1.5. Figure 1.9 shows a snapshot of the built-in file manager of RStudio with unfolded the pull-down option More. You can see that directly from within RStudio you can:

1. Create a New Folder
2. Create a New Blank File (not visible in the figure, but available in newer versions of RStudio)
3. Delete selected files
4. Rename selected files
5. In the pull-down option More, a.o.:
    a. Copy files
    b. Move files
    c. Set the directory currently visible in the built-in file manager as current working directory
    d. Display the working directory in the built-in file manager
    e. Show the currently opened folder of the built-in file manager in the file manager of your operating system

The ... in the right side of the built-in file manager allows you to select a different drive, e.g. a pen drive you plugged into a USB port of the computer or an external hard disk drive.

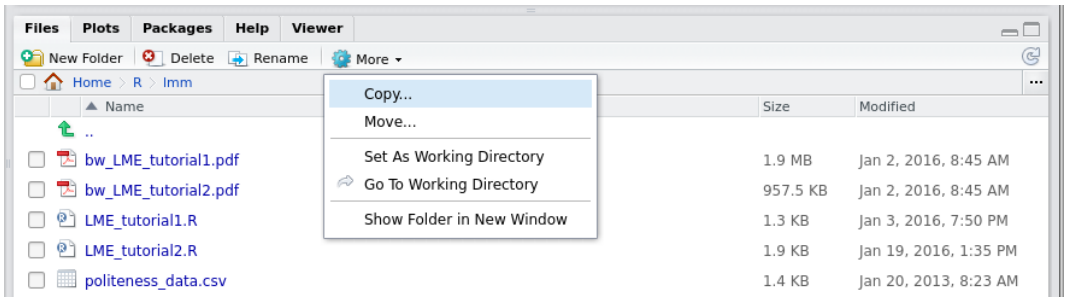


Figure 1.9: RStudio file manager with unfolded More options.

To be able to understand how navigating folders and files in R works, let's assume that the course files from Brightspace have been extracted in `~/R/data`. Suppose the current working directory, as shown in Figure 1.8, is `~/R`. In Windows this will refer to `C:/Users/<YOUR_USERNAME>/Documents/R`, whereas in macOS and Linux this will, respectively, refer to `/Users/<YOUR_USERNAME>/R` and `/home/<YOUR-USERNAME>/R`.

When executing the following command:

```
FoodDrinks <- read.csv(file = "./FoodDrinks.csv")
```

, the result will be an error message. The `./` in the `file` argument of the *read.csv()* function is used to indicate the current working directory. The command, you are executing, tries to load data contained in a comma separated value (CSV) file into a R object named ***FoodDrinks***. What, do you think, is the reason for this error?

Try it again by specifying the path to the file correctly. Remember that the file, you are trying to read into R, is located in `~/R/data`, a sub-directory of your current working directory.

Before continuing remove the object ***FoodDrinks*** from your workspace environment with the command:

```
rm(FoodDrinks)
```

Check on the Environment tab situated in panel 3 as displayed in Figure 1.5, that the object is not in your workspace environment anymore. Alternatively you can execute the command to list all objects currently in your workspace environment in the console:

```
ls()
# or equivalently
objects()
```

Another way of fixing this error is by changing the current working directory to where the file is located. You can do this in the built-in file manager by navigating to the `~/R/data` directory and from the pull-down option More select "Set As Working Directory". Use the *getwd()* function in the console to verify that the working directory has changed. Now retry the command, that resulted in an error before.

If you have changed your current working directory correctly the object ***FoodDrinks*** will be created, as you can verify with the *ls()* or *objects()* function command in the console or by looking at the Environment tab in RStudio.

This clearly demonstrates that commands, especially for loading data, are executed in the R console relative to the current working directory.

To prevent you from getting Complaints of Arm, Neck and/or Shoulders (CANS, formerly known as RSI or Repetitive Strain Injury) from excessive mouse use, setting the current working directory could also be achieved by executing this console command:

```
setwd("./data")
```

, where the assumption is that the current working directory is `~/R` and that the `data` directory exists as a sub-directory on that specific path.

In R commands you can use `..` to move up a directory. Execution of the command:

```
setwd("../../")
```

, will bring you back to the HOME directory represented by `~`. Execute the `setwd("../../")` command and verify with *getwd()* in the console, that your working directory has moved up two levels. You can of course also use the built-in file manager again. However, you will notice that the more advanced you will get in R, the more you will be using console commands instead of navigating with the mouse cursor and clicking the options.

The default working directory, the one RStudio displays at start up, can be set in the Global options..., situated under Tools in the top menu bar of RStudio. It is smart to set your default working directory to `~/Documents/R`, provided that this directory exists. You may have to create it first e.g. with your favorite file manager or within R. This will ensure, that all your R work will be saved in that specific directory. Of course, you should also create sub-directories there as well to keep your files organized.

Now at the end of this exercise clean up your workspace environment by executing:

```
rm(list = ls())
# or equivalently
rm(list = objects())
```

> Tip: Extend your knowledge about Workspace and Files
>
> To further extend your knowledge about the workspace and files, you could do SWIRL Lesson 2 at home.
>
> In this lesson you will learn more about directory and file management via commands in the R console.

1.6.3 Exercise SWIRL: Sequences of Numbers

In this exercise you will learn how to generate sequences in R. If you do not have the **swirl** package loaded, remember to first execute:

```
library(swirl)
```

Start swirl by executing:

```
swirl()
```

Now select SWIRL Lesson 3: **Sequences of Numbers** and work through the examples and questions.

1.7 Descriptive statistics and t-Test

For theory see O&L Sections 3.6, 4.9, 4.10, 5.2, 5.4 (in both 6^{th} and 7^{th} Ed. [4,5]).

1.7.1 Exercise Descriptive statistics: Area of Cultivation

A researcher did a small study about cultivation under glass. He asked 16 growers the area of cultivation under glass (given in Table 1.3).

Table 1.3: Measured areas of cultivation

Area of Cultivation							
53	30	44	39	39	25	73	60
22	98	30	24	49	40	36	50

a. Make a vector object, called ***area***, in R with the data given in Table 1.3.
b. Make a histogram of these data, by executing `hist(area)`.
c. Calculate the mean, median, the range, the first and third quartile by executing `mean(area)`, `range(area)`, and `quantile(area)`.

> **Note: Quantiles in R**
>
> Notice that the quantiles calculated in R appear to be on a $n-1$ basis, where n represents the number of observations. This is, however, not fully true, it just looks like it in this exercise.
>
> When calculating quantiles on a $n-1$ basis, the ordered observations are labeled starting with 0. The first quantile is then calculated as the $0.25 \times (n-1)$-th ordered observation. This is the same as the QUARTILE.INC function in Microsoft Excel. SPSS on the other hand uses by default a $n+1$ basis, which has as equivalent in Microsoft Excel the QUARTILE.EXC function.
>
> When calculating quantiles on a $n+1$ basis the first ordered observation is labeled 1 instead of 0 and, therefore, leading to different results. In order to get quantiles in R on a $n+1$ or $n-1$ basis you should execute the *quantile()* function with an additional argument:
>
> ```
> quantile(area, type = 6) # quantiles on a n + 1 basis
> quantile(area, type = 7) # quantiles on a n - 1 basis
> ```

d. Ask for summary statistics by execution of `summary(area)`.
e. Make a boxplot of the data with the following command: `boxplot(area)`.
f. Compare the boxplot with the results obtained in c.
g. Calculate the variance and standard deviation, by using the `var(area)` and `sd(area)` functions.
h. Calculate the standard error of the mean, `sd(area) / sqrt(n)`, and assign it to an object named ***SEmean***. Remind yourself that the number of observations can be obtained via `n <- length(area)`. Hint: Execute the commands in the proper order!

To calculate the 95% Confidence Interval of the mean in R you have to start with obtaining the correct t-distribution quantile value. In R all t-distribution quantile values can be obtained using the *qt()* function,

with proper values for its arguments. Look at the help of *qt()* to read what the arguments mean. To obtain the correct *t*-distribution quantile value for this exercise execute:

```
qtvalue <- qt(p = 0.025, df = 15, lower.tail = FALSE)
```

i. Check the value of ***qtvalue*** with a print command. Why are there `df = 15`, and `p = 0.025` in this *qt()* function?

Calculate the 95% Confidence Interval of the mean by executing the following commands:

```
lower <- mean(area) - qtvalue * SEmean
upper <- mean(area) + qtvalue * SEmean
print(lower)
print(upper)
```

j. What is the meaning of this 95% Confidence Interval?

The 95% Confidence interval can also be obtained via:

```
t.test(area)
```

k. Try it and compare the result with your given answers.

1.8 Basics in R . . . continued

1.8.1 Exercise SWIRL: Vectors

In R everything is an object and every object has a class. R has 5 basic atomic classes:

- logical (e.g. `TRUE` or `FALSE`)
- integer (e.g. `3L` or `as.integer(3)`)
- numeric (real or decimal, e.g. `2`, `2.0`, `pi`)
- complex (e.g. `1+0i`, `1+4i`)
- character (e.g. `"a"`, `"name"`)

The *class()* function can be used on an object to see, to which class the object belongs. R also has many data structures. These include among others:

- vector
- list
- matrix
- data frame
- factor
- array

In SWIRL lesson 4 the data structure vector is introduced. A vector is the most common and basic data structure in R, and is pretty much the workhorse of R. Vectors can be of two types:

- atomic vectors, which can only contain objects of one atomic class
- lists, which can contain objects of multiple atomic classes

When different atomic classes are mixed in an atomic vector, *coercion* occurs so that every element in the vector is of the same class. In R coercion always takes places to the most flexible atomic class (from the least to most flexible: logical, integer, numeric (or double), and character). Try the following examples yourself:

```
y <- c(1.7, "a")
class(y)
```

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

```
print(y)
```

```
#> [1] "1.7" "a"
```

```
y <- c(TRUE, 2)
class(y)
```

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

```
print(y)
```

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

```
y <- c("a", TRUE)
class(y)
```

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

```
print(y)
```

```
#> [1] "a"    "TRUE"
```

Objects can be explicitly coerced from one class to another using the *as.** functions, if available. Nonsensical coercion results in `NA`'s. Here are some examples you can also try yourself:

```
x <- 0:6
class(x)
```

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

```
as.numeric(x)
```

```
#> [1] 0 1 2 3 4 5 6
```

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

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

```
as.character(x)
```

```
#> [1] "0" "1" "2" "3" "4" "5" "6"
```

R objects can have attributes:

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class
- length
- other user-defined attributes / metadata

Attributes of an object can be accessed using the *attributes()* function.

Lists are a special type of vector, that can contain elements of different classes. Lists are a very important data type in R, and you should get to know them well.

```
x <- list(value = 1, char = "a", logi = TRUE, complex = 1 + 4i)
class(x)
```

```
#> [1] "list"
```

```
print(x)
```

```
#> $value
#> [1] 1
#>
#> $char
#> [1] "a"
#>
#> $logi
#> [1] TRUE
#>
#> $complex
```

```
#> [1] 1+4i
```

Start SWIRL and work on Lesson 4, entitled **Vectors**, to learn about this data structure and its use.

1.9 Command Reference

Table 1.4: Lesson 1 commands, operators and constants

Command	Description
`:`	colon operator to create regular sequences
abs()	absolute value
args()	function argument list
attributes()	retrieve or assign attributes of an object
boxplot()	create box-and-whisker plot of data
c()	combine values into a Vector or List
class()	retrieve the class of an object
colSums()	column sum
date()	system date
exp()	computes exponential function
factorial()	compute mathematical factorial
getwd()	get working directory
help() or `?`	primary interface to help in R
help.search()	search the help system in R
hist()	create a histogram
install.packages()	install packages in R
length()	length of an object
`letters`	built-in constant lower-case Roman alphabet
`LETTERS`	built-in constant upper-case Roman alphabet
library()	load and attach add-on packages
log()	natural logarithm
log10()	common base 10 logarithms
ls()	list objects
mean()	calculate (trimmed) arithmetic mean
objects()	list objects (equivalent to *ls()*)
paste()	concatenate strings
`pi`	built-in mathematical constant π
print()	print values to console
q()	quit/terminate an R session
qt()	compute a *t*-dist. quantile value for a given curve area and df
quantile()	calculate sample quantiles
range()	determine the range (minimum and maximum)
read.csv()	read a `*.csv`-file into a table format
rep()	replicate elements of vectors / lists
rm()	remove objects from workspace environment
rnorm()	random number generation from normal distribution
round()	rounding of numbers
scan()	read data into a vector or list
sd()	compute the sample standard deviation
setwd()	set working directory
seq()	sequence generation
seq_along()	sequence generation
seq.int()	sequence generation
seq_len()	sequence generation
sqrt()	compute the (principal) square root
summary()	summarize results of data or model fitting functions
t.test()	perform *t*-tests on data
var()	compute the sample variance

Command	Description

Table 1.5: Logical Operators

Command	Description
`&` or `&&`	logical and / intersection
`|` or `||`	logical or / union
xor()	logical exclusive or
`!`	logical negation (NOT)

Table 1.6: Relational Operators

Command	Description
`<`	less than
`>`	greater than
`==`	logical equality
`<=`	less than or equal to
`>=`	greater than or equal to
`!=`	not equal to

Lesson 2

Missing Values, Subsetting Vectors, Logic, and One- and Two-sample t-Tests

2.1 Missing Values, Subsetting Vectors and Logic

2.1.1 Missing Values

In R missing values are represented by the symbol `NA` (a.k.a. Not Assigned or Not Available). Undefined mathematical operations (e.g., divisions by zero) are represented by the symbol `NaN` (a.k.a. Not a Number). R uses the same symbol for data of class character and data of class numeric.

2.1.1.1 Testing for missing values

The *is.na()* function is used to test for `NA` values in an object and returns a logical vector with the same length as the tested object, where `TRUE` indicates `NA` and `FALSE` indicates non-`NA` values. To test for `NaN` values in an object, the *is.nan()* function can be used in the same way as described for *is.na()*.

`NA` values also have a class, so there are integer `NA` , character `NA`, *etc.* A `NaN` value is also `NA`, but the opposite is not true (not every `NA` is a `NaN`).

2.1.1.2 Excluding missing values from analyses

Many functions in R have an argument, which allows for exclusion of missing values. An example is *mean()*. Try the following commands yourself:

```
x <- c(1, 2, NA, 10, 3)
mean(x)
```

```
#> [1] NA
```

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

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

There are many other functions to exclude missing values from your analyses, some of them you will come across in this course, for example by using *is.na()* and *is.nan()* introduced in Section 2.1.1.1.

2.1.2 Subsetting Vectors

Subsetting is important and often used in statistics and data analysis. Not always analyses are run over the full set. Sometimes the data sets are split into training and validation sets, where the training set is used to build your statistical model. Subsetting of a vector is achieved by placing an index vector between square brackets after the object name of the vector to be subsetted, e.g. `x[1:2]` for the first two positions of `x`. Within R four types of index vectors can be used:

- logical vectors

- vectors of positive integers
- vectors of negative integers
- vectors of character strings

In the following example the first four elements of a character vector are taken:

```
x <- c("a", "b", "c", "c", "d", "a")
class(x)
```

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

```
x[1:4]
```

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

This example uses a vector of positive integers to subset the given character vector.

In SWIRL Lesson 6: **Subsetting Vectors**, which is one of the exercises, you will learn about all possible index vectors you can use for subsetting a vector.

2.1.3 Logic

Logical vectors can be used for subsetting vectors. The two logical values in R, also called Boolean values, are `TRUE` and `FALSE` (write them with all capitalized letters in R!). Apart from subsetting with logical vectors, relational and logical operators can be used in R to construct logical expressions, which will evaluate to either `TRUE` or `FALSE`. These are useful when building functions later on in this course to evaluate certain conditions.

The first relational operator is the equality operator, represented by two equals signs `==`. For example:

```
TRUE == TRUE
```

should evaluate to `TRUE`.

Just like arithmetic, parentheses can be used to set the priority of evaluation, e.g.

```
(FALSE == TRUE) == FALSE
```

What do you think the outcome of this logical expression will be? Execute it to check your answer.

The equality operator can be used to check two objects of the same class for equality, where it does not matter which class as long as they are the same. The following example should evaluate to `FALSE`:

```
6 == 7
```

Thankfully relational operators exist, that allow testing for a value less than or greater than another value as well. The less than operator `<` tests, whether the number on the left-hand side of the operator (called the left operand) is less than the number on the right-hand side of the operator (the right operand). There is also a less-than-or-equal-to operator `<=`. Equivalently there is a greater than `>` and a greater-than-or-equal-to `>=` operator.

The inequality operator is represented by `!=`, which can be read as 'not' displayed by `!` and 'equal' represented by `=`. The exclamation point in R is, what is called the negation operator, basically meaning NOT. In order to negate logical expressions you can use the NOT operator. An exclamation point `!` will cause `!TRUE` (say: not true) to evaluate to `FALSE` and `!FALSE` (say: not false) to evaluate to `TRUE`.

At some point the need will arise to examine relationships between multiple expressions. This is where the **AND** operator, and the **OR** operator come into play. First let us scrutinize the operation of **AND**. There are two AND operators in R, `&` and `&&`. Both operators work similarly, if the right and left operands of **AND** are both `TRUE` the entire expression evaluates to `TRUE`, otherwise it is `FALSE`. The `&` operator can be used to evaluate **AND** across a vector. The `&&` version of **AND** only evaluates single logical outcomes on the left, and right hand side of the operator with each other. A few examples:

```
FALSE & FALSE
TRUE & c(TRUE, FALSE, FALSE)
# and
TRUE && c(TRUE, FALSE, FALSE) # Try also TRUE && TRUE
```

In order to truly get what is happening, try to figure out the answer yourself before executing these commands in the console.

A similar set of rules is followed by the **OR** operator. The | version of **OR** evaluates across an entire vector, while the || version of **OR** only evaluates the first member of a vector. An expression using the **OR** operator will evaluate to TRUE if the left operand or the right operand is TRUE. If both are TRUE, the expression will result TRUE, however if neither are TRUE (or equivalently both are FALSE), then the expression will be FALSE. Two examples:

```
TRUE | c(TRUE, FALSE, FALSE)
# and
TRUE || c(TRUE, FALSE, FALSE)
```

Again try to figure out the outcome, before trying it in the console.

The power of logical operators lies in the fact that they can be chained together just like arithmetic operators. The expressions:

```
6 != 10 && FALSE && 1 >= 2
# or
TRUE || 5 < 9.3 || FALSE
```

are perfectly normal in R. Just keep in mind that logical operators have an order of operation, just as arithmetic operators.

> ! Important: Evaluation order of AND and OR operators
>
> **All AND operators are evaluated before OR operators.**
>
> Therefore, use parentheses to set precedence in the evaluation order.

Try to figure out the following example:

```
5 > 8 || 6 != 8 && 4 > 3.9
```

This expression will evaluate to TRUE. First the left and right operands of the **AND** operator are evaluated. 6 is not equal 8, 4 is greater than 3.9, therefore both operands are TRUE so the resulting expression TRUE && TRUE evaluates to TRUE. Then the left operand of the **OR** operator is evaluated: 5 is not greater than 8 so the entire expression is reduced to FALSE || TRUE. Since the right operand of this expression is TRUE the entire expression evaluates to TRUE.

2.1.3.1 R functions for dealing with logical expressions

Having familiarized with R's logical operators a few functions will be introduced for dealing with logical expressions.

The function *isTRUE()* takes one argument. If that argument is TRUE, the function will return TRUE, otherwise it returns FALSE.

The function *identical()* will return TRUE if the two R objects passed to its arguments are identical. For example:

```
identical("twins", "twins")
```

Another quite useful function is the *xor()*, which takes two arguments. The *xor()* stands for exclusive **OR**. This function only evaluates to TRUE, if and only if just one of the two arguments is TRUE. When both arguments are TRUE the function evaluates to FALSE, hence the exclusivity.

The *which()* function takes a logical vector as an argument and returns the indices of the vector that are TRUE. For example:

```
which(c(TRUE, FALSE, TRUE))
```

would return the vector c(1, 3). Like the *which()* function, the functions *any()* and *all()* take logical vectors as their argument. The *any()* function will return TRUE, when one or more of the elements in the logical vector is or are TRUE. The *all()* function will return TRUE, when every element in the logical vector is TRUE.

2.1.4 Exercise SWIRL: Missing Values

Start *swirl()* from the package **swirl** and work through lesson 5 about **Missing Values** in R. If you do not remember how to start up SWIRL, have a glimpse at Section 1.6.3.

2.1.5 Exercise SWIRL: Subsetting Vectors

Start up SWIRL and work through lesson 6, where you will learn more about **Subsetting Vectors** as announced in Section 2.1.2.

2.2 One- and Two-sample t-Tests in R

For theory see O&L Sections 6.2, 6.4 (in both 6^{th} and 7^{th} Ed. [4,5]).

2.2.1 Exercise Descriptive Statistics and t-Test: Tasting Coffee

In a test it was investigated whether consumers are more positive or negative about a certain brand of coffee in a test situation at home compared to tasting the coffee in a laboratory situation. The consumers were asked to score their judgment about the coffee by moving an arrow on a vertical axis with numbers between 0 and 100. You may assume that the differences between the two scores per consumer are normally distributed. The results of 10 randomly selected consumers are given in Table 2.1.

Table 2.1: Coffee tasting data.

Consumer	1	2	3	4	5	6	7	8	9	10
Judgement at home	50	76	81	60	30	70	74	64	76	70
Judgement in the lab	55	74	79	49	34	65	68	66	73	64

These data are also available in the file named `tasting_coffee.csv` within the course data file you downloaded from Brightspace and extracted into your folder `~/R/Data`.

a. Assuming that your default working directory has been set to `~/R`, load the data for this exercise by executing in the console:

```
judgmentData <- read.csv(file = "./data/tasting_coffee.csv")
```

> **Note: Eye-ball (inspect) the data**
>
> You can inspect the data by printing the object (autoprinting or explicit print command) or by executing `View(judgmentData)` in the console of RStudio.

> **Important: Transform into vectors**
>
> The result might surprise you. Instead of three vector objects, named ***Consumer***, ***home*** and ***lab***, we obtain ***judgmentData*** with '10 observations of 3 variables', a so called data frame (which will be discussed in more detail in lesson 3).
>
> Since the use of a `data.frame` object has not been discussed yet, let's for now transform this data frame into vector objects by executing, sequentially, the following three commands in the console:
>
> ```
> consumer <- judgmentData$Consumer
> home <- judgmentData$home
> lab  <- judgmentData$lab
> ```

b. Make a histogram of the judgments at home and ask for summary statistics.

c. Also make a histogram of the judgments in the lab, ask for summary statistics and compare with the result in b.

d. Since the data are paired (a consumer gives a judgment both at home and in a lab) calculate the difference between the two judgments (and call the new vector ***d***).

e. Make a boxplot of ***d***. Is the assumption that the differences between the two scores per consumer are normally distributed reasonable?

f. What is the best estimate of the population mean difference in judgment?

g. How precise is this estimate, i.e. what is the 95% confidence interval for the mean difference?

h. What is the meaning of this 95% confidence interval? So, what can you conclude about the difference in judgment at home as compared to judgment in the lab?

Tip: Check your answer

A fast way to check your answers is to use *t.test()*. Look at the help of *t.test()* and try to figure out which function arguments to use!

2.2.2 Exercise: Comparing samples from Normal Distributions

a. Draw a sample of size 100 from a normal distribution, with mean of 50 and a standard deviation of 5, and assign it to an object named ***x1***.

Note

Looking at the help page for *rnorm()* (executing ?rnorm' in the console) you can read, which arguments you have to specify to obtain the desired response that creates ***x1***. It should be looking like this:

```
x1 <-  rnorm(n = 100, mean = 50, sd = 5)
```

b. Make a histogram of the resulting object ***x1***.

c. Calculate the mean and standard deviation of the sample. Are they what you expected (close to the given mean and standard deviation)?

d. Test in a one-sample t-test if the mean is different from (the expected) 50.

Note

Reading the help page (`?t.test`) shows that a One-Sample t-test can be performed with:

```
t.test(x = x1, mu = 50) # alternative = "two.sided" [by default]
```

e. Draw another sample (assign to ***x2***) from a normal distribution but now with `mean = 53` and `sd = 5`. Again, make a histogram. What is the difference? What changes in the shape of the histogram?

f. Compare the two means using a t-test, in other words test $H_0: \mu_1 = \mu_2$ against the alternative that the means differ. Here you can assume equal variances. What is the conclusion?

Note: Independent Samples T-test and Levene's Test for Homogeneity of variances

You can do an Independent Samples t-test, with the assumption of equal variances, in R with the following command:

```
t.test(x1, x2, var.equal = TRUE)
```

By default the Independent Samples t-test is performed in R with the assumption of unequal variances, and R uses the Welch-Satterthwaite approximation to determine the degrees of freedom.

Levene's Test for Homogeneity of variances can be performed in R, but this test is not by default available as R function. You can extend the functionality of R with the **car** package [6] and perform Levene's Test by executing:

```
# Load the car package
library(car)
# Group object x1 and x2 in object y
y <- c(x1, x2)
# Create a grouping variable named group
group <- as.factor(c(rep(1, length(x1)), rep(2, length(x2))))
# Perform Levene's Test for Homogeneity of Variance
leveneTest(y = y, group = group, center = mean)
# Unload the car package
detach("package:car", unload = TRUE)
```

From the output of Levene's Test you can conclude, whether the Independent Samples t-test should be performed assuming equal variances or not.

g. Now we are going to investigate what would happen if the sample size is much smaller. So, we only want to use the first 12 values of ***x1*** and ***x2***. Make a subset of ***x1*** and ***x2*** of the first 12 values and call the subsets ***x3*** and, respectively, ***x4***.

h. Perform a test to compare the mean of ***x3*** with ***x4***, in other words test the hypothesis H_0: $\mu_3 = \mu_4$ against the alternative that the means differ (again assume equal variances). What is the conclusion?

i. Compare the result of the test in f. with result in h. and explain the difference.

2.2.3 Exercise: Two-sample t-Test

A full description of this problem can be found in O&L 6th Edition pp. 292-293, and 325-330 [4] or O&L 7th Edition pp. 302-303, and 336-341 [5].

On January 7, 1992, an underground oil pipeline broke and caused the contamination of a marsh along the Chiltipin Creek in Texas, USA. The cleanup process consisted of burning the contaminated regions in the marsh. To evaluate the influence of the oil spill on the mean flora density (μ), researchers designed a study of plant growth 1 year after the burning. The data collected consists of 40 measurements of flora density in the uncontaminated (1 = control) sites and 40 density measurements in the contaminated (2 = burned) regions.

The data are available in the file `oil_spill.csv`.

a. Read in the data with the *read.csv()* function and assign it to an R object named ***oilspill***. If you don't remember how to do this, check Section 2.2.1. Transform the data frame (which will be handled in more detail tomorrow) to vectors by executing sequentially: `tract <- oilspill$tract`, `density <- oilspill$density`, and `site <- oilspill$site`.

b. Make a boxplot of the density in the control sites and the oil-spill sites separately using the following command: `boxplot(density ~ site, col = "khaki")`. Ask for overall summary statistics.

> **i** Note: Use of ~ in a formula
>
> In the previous question you have created side-by-side boxplots for density in the control and oil-spill sites. To do this you have specified a formula using the tilde operator, written in R with ~. You can read this operator as **versus** in a graphical representation. So basically you have asked R to make side-by-side boxplots of density versus site.
>
> The `col` function argument, as specified in the command, can be used to specify the color of the box in the boxplot.

c. Ask for summary statistics for each site separately. First make two objects (***dens_control*** and ***dens_oilspill***) by means of subsetting in an appropriate way.

d. Carry out the t-test for H_0: $\mu_1 \leq \mu_2$ against H_a: $\mu_1 > \mu_2$ at $\alpha = 0.05$. Do not forget to first check with Levene's Test, `leveneTest(y = density, group = as.factor(site), center = mean)`, whether you can assume variances to be equal or not. Give the outcome of the test-statistic, the p-value for the

one-sided problem and the conclusion of the test (in words). Check the help (`?t.test`) on how to fill in the function arguments.

e. Determine the 0.95 Confidence Interval for $\mu_1 - \mu_2$. Which conclusion follows for the test with H_a: $\mu_1 - \mu_2 \neq 0$?

f. Another way of performing the t-test is with: `t.test(density ~ site)`. Do this and compare the output of d. with the one just generated. Is there difference, and if so, what is the difference?

2.3 Additional SWIRL Exercises

If you are finished with the statistical exercises before today's lesson ends, please do the following exercises.

2.3.1 Exercise SWIRL: Workspace and Files

Start *swirl()* from the package **swirl** and work through lesson 2 about **Workspace and Files** in R. In this lesson you will learn more on how to deal with the workspace and do file management from the console.

2.3.2 Exercise SWIRL: Logic

Start *swirl()* from the package **swirl** and work through lesson 8 about **Logic** in R.

2.4 Command Reference

Table 2.2: Lesson 2 commands, operators and constants

Command	Description
~	tilde operator used to separate the left- and right-hand sides in a model formula
all()	check if all values are `TRUE`
any()	check if some values are `TRUE`
detach()	detach object, o.a. package, from the search path
identical()	test objects for exact equality
is.na()	check for 'Not Available' / Missing Values
is.nan()	check for 'Not a Number' / result of invalid math operation
isTRUE()	check if object equals `TRUE`
leveneTest()	test for homogeneity of variance across groups (**car** package)
names()	get or set the names of an object
View()	Invoke RStudio's data viewer
which()	check which indices are `TRUE`

Table 2.3: Logical Operators

Command	Description
& or &&	logical and / intersection
| or ||	logical or / union
xor()	logical exclusive or
!	logical negation (NOT)

Table 2.4: Relational Operators

Command	Description
<	less than
>	greater than

Command	Description
`==`	logical equality
`<=`	less than or equal to
`>=`	greater than or equal to
`!=`	not equal to

Lesson 3

Matrices, Data frames, Factors and Binomial Tests

3.1 Matrices, Data frames and Factors

Matrices and Data frames are both used to store data in a tabular form with rows and columns, and both data types, therefore, represent 'rectangular' data types. There is, however, one big difference between the two. Matrices can only contain one single class of data, whereas data frames can consist of many different classes of data.

Factors are used in R to encode categorical data in a vector.

3.1.1 Matrices

Matrices in R are vectors with a *dimension* attribute. The dimension attribute is itself, for a matrix, an integer vector of length 2, where the first integer specifies the number of rows and the second integer specifies the number of columns. For example:

```
myVector <- 1:12
dim(myVector) <- c(3, 4)
```

When you print ***myVector*** in the console you will see that it has been transformed into matrix. To check you can ask for the object class of ***myVector***, which will show that ***myVector*** has class "*matrix*". Apart from setting the *dimension* attribute with the *dim()* function it can also be used to retrieve the dimensions of an R object. Alternatively you can use the *attributes()* function, which will show the dimension attribute if present.

Instead of setting the dimension attribute of a vector, matrices can also be constructed directly with the *matrix()* function. Matrices are constructed column-wise, so entries can be thought of starting in the "upper left" corner and running down the columns. The matrix created from ***myVector*** could for example be constructed directly:

```
myMatrix <- matrix(data = 1:12, nrow = 3, ncol = 4)
```

A third way to create matrices is by row-binding or column-binding, which can be achieved with the *rbind()* and *cbind()* functions. As example consider:

```
x <- 1:3
y <- 10:12
cbind(x, y)
```

```
#>      x  y
#> [1,] 1 10
#> [2,] 2 11
#> [3,] 3 12
```

```
rbind(x, y)
```

```
#>   [,1] [,2] [,3]
```

```
#> x    1    2    3
#> y   10   11   12
```

The *rbind()* function allows, as the example shows, for binding/attaching data row-wise, whereas the *cbind()* function allows for column-wise binding/attachment of data.

Multi-way arrays are similar to matrices but can have more than two dimensions. To create a $3 \times 3 \times 3$ multi-way array containing the values 1:27 execute:

```
myArray <- array(data = 1:27, dim = c(3, 3, 3))
```

3.1.2 Data frames

A **data frame** in R is used to store tabular data (also known as a data table) and represents a special type of list, where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list equals the number of rows. Unlike matrices, as mentioned at the beginning of this lesson, data frames can store different classes of objects in each column (just like lists); matrices must have every element of the same class. Here is an example of a data frame object named ***df***:

```
df <- data.frame(user = 1:4, access = c(TRUE, TRUE, FALSE, FALSE))
```

> **i** Note: Write Boolean values in full.
>
> Notice here that `TRUE` or `FALSE` have been written out in full , instead abbreviations `T` and `F` can also be used for the basic logical constants (even though this is generally considered bad practice, as you will learn in Lesson 6).

A data frame has dimensions, just like a matrix. To print the dimensions of a data frame to the console you can use the same command as for a matrix, being the *dim()* function. The dimensions of the ***df*** object are:

```
dim(df)
```

```
#> [1] 4 2
```

You could also use the *nrow()* and *ncol()* functions to specifically obtain the number of rows or columns of a data frame, e.g. for object ***df***: `nrow(df)` and `ncol(df)`.

Data frames also have a special attribute called `row.names`, which is a character vector of length the number of rows with no duplicates nor missing values. The function *row.names()* can be used to get and set row names for data frames. The `names` attribute of a data frame specifies the column names, and these can be retrieved or set with the *names()* or *colnames()* functions. For example using the previously defined ***df*** object:

```
row.names(df) <- c("Peter", "Ingrid", "Harry", "Sally")
print(df)
```

```
#>        user access
#> Peter     1   TRUE
#> Ingrid    2   TRUE
#> Harry     3  FALSE
#> Sally     4  FALSE
```

The resulting output from executing the *attributes()* function on the ***df*** object:

```
attributes(df)
```

```
#> $names
#> [1] "user"   "access"
#>
#> $class
#> [1] "data.frame"
#>
#> $row.names
#> [1] "Peter"  "Ingrid" "Harry"  "Sally"
```

Data frames are usually created by calling *read.table()*, or its version specially made for comma separated value files *read.csv()*, to read in data from an external file. Data frames can be converted to a matrix by calling *data.matrix()*, but be aware of coercion since a matrix can only hold one single class of data.

3.1.3 Factors

As stated at the beginning of this lesson factors are used to represent categorical data. Factors can be unordered (nominal variable) or ordered (ordinal variable). You can think of a factor as an integer vector, where each integer has a label.

Factors are treated specially by modeling functions, e.g. *lm()*, used for fitting Linear Models, and *glm()*, which is used for fitting Generalized Linear Models. Using factors with labels is better than using integers, because factors are self-describing; having a variable that has for example values "Male" and "Female" is better than a variable that has values 1 and 2. An example on how to create an object ***gender*** of class *"factor"*:

```
(gender <- factor(c(1, 1, 2, 1, 2, 2, 1, 2), labels = c("Male", "Female")))
```

```
#> [1] Male   Male   Female Male   Female Female Male   Female
#> Levels: Male Female
```

```
class(gender)
```

```
#> [1] "factor"
```

or alternatively:

```
gender <- factor(c("Male", "Male", "Female", "Male", "Female", "Female", "Male", "Female"),
                 levels = c("Male", "Female"))
```

Here the `levels` argument is used to specify the order of the categorical data and in this example 1 will represent "Male" and 2 will represent "Female. Had the `levels` argument been specified as `levels = c("Female", "Male")` it would be the other way around (1 ="Female" and 2 = "Male").

The *levels()* function can be used on a factor object to display the order of the `levels` attribute of an object. For example:

```
levels(gender)
```

```
#> [1] "Male"   "Female"
```

So far the *factor()* function has been applied to create a nominal variable object. The `ordered` function argument can be used to create an ordinal variable object as well, where the order is assigned as in the supplied order. For example:

```
rating <- factor(c(1, 2, 1, 2, 3, 2, 3, 2, 3),
                 labels = c("small", "medium", "large"),
                 ordered = TRUE)
print(rating)
```

```
#> [1] small  medium small  medium large  medium large  medium large
#> Levels: small < medium < large
```

```
levels(rating)
```

```
#> [1] "small"  "medium" "large"
```

```
class(rating)
```

```
#> [1] "ordered" "factor"
```

Alternatively the same ordinal variable object ***rating*** can also be created with the *ordered()* function:

```
rating <- ordered(c(1, 2, 1, 2, 3, 2, 3, 2, 3), labels = c("small", "medium", "large"))
```

To obtain the integer values used for assigning the levels in the factor object, the *unclass()* function can be applied. For example:

```
unclass(rating)
```

```
#> [1] 1 2 1 2 3 2 3 2 3
#> attr(,"levels")
#> [1] "small"  "medium" "large"
```

With the *table()* function an overview of the number of observations of each level can be created. For example on our ***gender*** and ***rating*** objects this would result in:

```
table(gender)
```

```
#> gender
#>   Male Female
#>      4      4
```

```
table(rating)
```

```
#> rating
#>  small medium  large
#>      2      4      3
```

3.1.4 Subsetting

There are a number of operators, that can be used to extract subsets of R objects:

`[` always returns an object of the same class as the original and can be used to select more than one element.
`[[` is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
`$` is used to extract elements of a list or data frame by name, with the semantics similar to that of `[[`.

So far you have seen the use of the square brackets `[ ]` for subsetting a vector in Section 2.1.2.

3.1.4.1 Subsetting Matrices

Matrices can be subsetted in the usual way with (i, j) type indices. For example:

```
(myMatrix <- matrix(1:12, 3, 4))
```

```
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    4    7   10
#> [2,]    2    5    8   11
#> [3,]    3    6    9   12
```

```
myMatrix[2, 3] # select element row 2 column 3
```

```
#> [1] 8
```

```
myMatrix[2:3, 2:4] # select a matrix from my_matrix
```

```
#>      [,1] [,2] [,3]
#> [1,]    5    8   11
#> [2,]    6    9   12
```

Notice that the arguments of the *matrix()* function have not been named in this example. R fills them in order of appearance, where it assumes the order as specified in the help of the function. It is, however, good practice to specify argument names. This makes your function call, and later your program code, more readable for yourself and other users, that might want to reuse your code.

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1 x 1 matrix. This behavior can be turned off by setting `drop = FALSE`, as shown in the following example:

```
myMatrix[1, 2]
```

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

```
myMatrix[1, 2, drop = FALSE]
```

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

Similarly, subsetting a single row or a single column will result in a vector by default, not a matrix. The function argument `drop = FALSE` can be used to return a matrix instead.

When indices are missing, a full row or column will be selected from a matrix. For example:

```
myMatrix[3, ] # select the third row
```

```
#> [1]  3  6  9 12
```

```
myMatrix[, 4] # select the fourth column
```

```
#> [1] 10 11 12
```

Subsetting multi-way arrays works the same way as explained above, only the number of indices needs to be extended and match the number of dimensions in the multi-way array.

3.1.4.2 Subsetting Data frames

As stated in Section 3.1.2, data frames represent a special type of list, which emphasizes the importance of lists in R. Subsetting of data frames in R uses, therefore, the same operators as subsetting of lists. Here examples will be given for subsetting of a list and in an exercise you will have to apply these rules for subsetting a data frame.

Let's first create a very simple list:

```
(x <- list(foo = 1:4, bar = 0.6))
```

```
#> $foo
#> [1] 1 2 3 4
#>
#> $bar
#> [1] 0.6
```

```
class(x)
```

```
#> [1] "list"
```

```
names(x)
```

```
#> [1] "foo" "bar"
```

You can see that the list contains two parts. One labeled `foo` containing four integer values and one labeled `bar` containing only one numeric value. The beginning of Section 3.1.4 stated, that subsetting with the `[` operator returns an object of the same class. Here as an example for our list:

```
x[1]
```

```
#> $foo
#> [1] 1 2 3 4
```

```
class(x[1])
```

```
#> [1] "list"
```

Filling the name of the part of the list between single square brackets will give the same result as supplying an integer value, as shown here:

```
x["foo"]
```

```
#> $foo
#> [1] 1 2 3 4
```

To really access the values in the first part (`foo`) of the list double square brackets `[[` should be applied. For example:

```
x[[1]]
```

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

```
class(x[[1]])
```

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

Instead of using the double square brackets `[[`, the dollar sign `$` can be used. The advantage of the dollar sign is, that names can be used and RStudio will immediately show the possible options after typing the `$` sign in the console behind an object, when multiple options are available. For example to extract the value of the second part of the list, named `bar`, the following equivalent commands can be applied:

```
x[[2]]
```

```
#> [1] 0.6
```

```
# or equivalently
```

```
x$bar
```

```
#> [1] 0.6
```

When subsetting a list with double square brackets, instead of giving integer values between the brackets, you can also type the name of the part of the list between quotes. The following statement is, therefore, equivalent to the previous two, but you would have to know the name of the part of the list you want to subset.

```
x[["bar"]]
```

```
#> [1] 0.6
```

The big difference between the `[[` and the `$` operator is that the `[[` operator can be used with **computed** indices; `$` can only be used with literal names. For example in our simple list:

```
name <- "foo"
x[[name]] # computed index for "foo"
```

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

```
x$name # element name does not exist in list x
```

```
#> NULL
```

```
x$foo # element foo does exist in list x
```

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

The `[[` operator can take an integer sequence. This is extremely useful when subsetting nested elements of a list (or data frame), e.g.

```
y <- list(a = list(10, 12, 14), b = c(pi, 2.81))
y[[c(1, 3)]] # third value in the list of part a
```

```
#> [1] 14
```

```
# can equivalently be done via
# y$a[[3]], or y[[1]][[3]], or y[["a"]][[3]]
```

```
y[[c(2, 1)]] # first element in the list of part b
```

```
#> [1] 3.141593
```

Both the `[[` and the `$` operator allow partial matching of names. Of course being lazy, in the sense of not typing more code than necessary, is generally a good way to keep your code compact. However, be careful when using partial matching, because it can lead to unexpected results. This example shows the use of partial matching of names:

```
z <- list(aardvark = 1:5)
z$a # matches to aardvark
```

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

```
z[["a"]] # "a" does not exist
```

```
#> NULL
```

With the `[[` operator by default partial matching does not work as seen in the previous example. However, partial matching can be forced by changing the value of the `exact` argument:

```
z[["a", exact = FALSE]] # [default: exact = TRUE]
```

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

> ⚠ Warning: Partial Matching
>
> Although partial matching can be used in R coding, it is generally considered as **bad** coding practice. As stated it can lead to unexpected result.
>
> The recommendation, therefore, is not to use it!

3.1.4.3 Subsetting for removal of missing values (`NA`)

A common task is to remove missing values (as introduced in Section 2.1.1). Subsetting is an easy way to perform this task:

```
x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x)
x[!bad] # only non NA values
```

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

If you have multiple vectors, a matrix, or a data frame with `NA` values on various positions and you want to take a subset with no missing values at all, there is an R function *complete.cases()* that can do this for you. An example using two vectors:

```
x <- c( 1, NA, NA, 4, 5, 6)
y <- c("a", "b", NA, "d", NA, "f")
good <- complete.cases(x, y)
x[good]
```

```
#> [1] 1 4 6
```

```
y[good]
```

```
#> [1] "a" "d" "f"
```

The following example uses the ***airquality*** data frame from the base package **datasets** to subset complete cases:

```
dim(airquality)
```

```
#> [1] 153   6
```

```
head(airquality) # last six rows via tail(airquality) [default: n = 6L]
```

```
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 4    18     313 11.5   62     5   4
#> 5    NA      NA 14.3   56     5   5
#> 6    28      NA 14.9   66     5   6
```

```
good <- complete.cases(airquality)
head(airquality[good, ])
```

```
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 4    18     313 11.5   62     5   4
#> 7    23     299  8.6   65     5   7
#> 8    19      99 13.8   59     5   8
```

3.1.5 Exercise SWIRL: Matrices and Data Frames

Start *swirl()* from the package **swirl** and work through lesson 7 to learn about **Matrices and Data Frames**. If you do not remember anything about swirl, look at the intro of Section 1.6.1. However, if you only do not remember how to start up swirl, quickly look at Section 1.6.3.

3.1.6 Exercise: Subsetting the *scores* matrix

a. For this exercise first create the ***scores*** object by reading in the `student_scores.csv` from your course data directory and assigning it to the aforementioned object name. By default, the *read.csv()* function creates an object of class "*data.frame*". Executing the following command turns the object into an object of class "*matrix*":

```
scores <- as.matrix(scores)
```

b. Check the class, dimensions of the ***scores*** matrix and whether it has named dimensions? Checking for named dimensions can be done in several ways, e.g. printing the object. The easiest way is the *dimnames()* function, read the help page (`?dimnames`) for more info.

c. Subset the ***scores*** matrix for the first four consecutive columns by subsetting with a sequence.

d. Subset the ***scores*** matrix to only see the columns `id`, `read`, `write`. Try to do this in two ways (using numerical values and by using named columns). Use one of your solutions to assign it to an object names ***scores2***.

e. Create a subset from the original ***scores*** matrix by selecting the first six rows. You do not have to assign this subset to a particular object name, just print it in the console. Also try to create a subset showing the last six rows of the original ***scores*** matrix. [Hint: remember there is a *nrow()* function, which counts the number of rows in an object.]

> Tip: Displaying the first and last number of *data.frame* rows
>
> Subsetting for the first six rows is not so complicated, however, subsetting the last six rows requires you to write an expression.
>
> In R there are two convenient functions named *head()* and *tail()* that by default display the first, and respectively, the last six rows of an object. The function argument `n` can be changed in case you prefer more or fewer rows, e.g. `tail(scores, n = 8)` will display the last eight rows of the ***scores*** matrix.

f. Create a subset of the original ***scores*** matrix, where the column `id` can have the following values: `86`, `48`, `195`, `12`. For this question you should use value matching, which can be achieved with `%in%`, look at the help page `?'%in%'` for its use. [Hint: first write the logical expression to select the requested `id` numbers!]

g. Calculate the mean `read` score from the subsetted ***scores*** matrix containing the columns `id`, `female`, `race`, `ses` and `read`, where the `ses` column is equal to 3. First create a logical expression (with `TRUE/FALSE`) to select rows in the ***scores*** object, where `ses` equals 3. Next use this expression to subset the ***scores*** matrix for the appropriate rows and select only the columns `id`, `female`, `race`, `ses` and `read` (column numbers 1:4 and 7). The resulting matrix should contain the following `id` numbers: 86, 141, 95, 104, 76, 114 and

20 (7 rows). Finally, select only the `read` column and calculate the mean. [answer: mean `read` score = 58.42857].

3.1.7 Exercise: Subsetting the *mtcars* data.frame

In this exercise the ***mtcars*** object from the base **datasets** package will be used. The data in this object was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models). Read additional info about this object by executing `?mtcars` in the console.

a. Check that the ***mtcars*** is indeed of class "*data.frame*".

b. Have a look at the data (a.k.a. eyeballing) contained in the ***mtcars*** by:

- printing the object to the console (using *print()*, auto printing or enclosing the object in parentheses).
- using the *head()* and _tail()} functions.
- using the RStudio specific *View()* function on the object.
- using the *str()* on the object.

> Tip: Looking at the data (Eyeballing)
>
> Having a look at the data, prior to performing analyses, is always a good idea. It will give you a feeling about what is contained.
>
> R users generally prefer to use the *str()* function, which compactly displays the structure of an arbitrary R object. In this case it provides information about the object class, the number of observations and variables, the element (or variable) names contained in the *data.frame* object and their class, as well as the first values of each element.
>
> The *head()*, *tail()*, and the RStudio specific *View()* function also provide nice overviews. You will understand the drawbacks of the *head()* and *tail()* functions, when there are lots of columns in your data causing your output in the console to spread out over multiple lines. The function *View()* has the similar drawback with respect to the number of columns, although RStudio allows for scrolling. The biggest drawback of the *View()* function is, that it is limited in the number of rows it can display (although not a problem for the small ***mtcars*** object).

c. Subset the ***mtcars*** object for only the number of gears (`gear` element / variable). Use the function *as.factor()* over this selection to turn the subset into a factor. What are the possibilities for the number of gears in the data.frame? [Hint: remember the *levels()* function or the attributes of a factor object]

d. Correct the following command, with two mistakes, for calculating the mean mileage per gallon (`mpg` element) for cars in the ***mtcars*** object with automatic transmission (`am` element equal to 0):

```
mean(mtcars[mtcars$am = 0]$mpg)
```

e. How many cars are there in the ***mtcars*** object with manual transmission (`am` element equal to 1) and 1 or 2 carburetors (`carb` element equal to 1 or 2)? Give their names and calculate the mean and standard deviation of the gross horsepower (`hp` element) of this subset. [Hint: look at the help pages `?nrow` and `?rownames`]

f. Fix each of the following common data frame subsetting errors:

- `mtcars[-1:5, ]`, to delete the first five rows
- `mtcars[mtcars$cyl <= 5]`, to select cars with the number of cylinders less than or equal to 5
- `mtcars[mtcars$cyl = 4 | 6, ]`, to select cars with the number of cylinders equal to 4 or 6

> Tip: Functions to aid subsetting
>
> When subsetting objects in R there are two very useful **base** package functions you can and, probably, will want to use. The first one is the *with()* function, which helps you reduce the amount you have to type. For example with the ***mtcars*** object of this exercise you can subset for the cars with exactly five

forward gears in the usual way:

```
mtcars[mtcars$gear == 5, ]
```

using the *with()* function this would look like:

```
mtcars[with(mtcars, gear == 5), ]
```

It might not look like an improvement, but consider a much longer conditional subsetting where each time you have to type the object name with an $ and a column selection in the logical expression.

The second function is the *subset()* function. Look at the help page of the function to read about its use, here is an example for two equivalent commands:

```
mtcars[mtcars$disp >= 300, ]
# equivalent with the subset() function
subset(mtcars, disp >= 300)
```

g. Create a boxplot for the weight (`wt` element) of cars in the ***mtcars*** data frame, with less than 5 gears (`gear` element) and the `vs` element equal to 0. In the *boxplot()* function add the following function arguments: `col = "lightblue"` (create a light-blue box), `ylab = "weight"` (add an y-axis label) and `main = "Boxplot of the conditional weight from mtcars"` (add a main title to the boxplot). Give an explanation about what happened to the upper whisker in this boxplot.

3.2 Binomial Tests

For theory see O&L Section 4.8 (in both 6^{th} and 7^{th} Ed. [4,5]), excluding the part about the Poisson Distribution.

3.2.1 Exercise Binomial Test: Experiment with the olfactometer

We have 100 silkworm moths that enter individually into a y-tube olfactometer. Each of the two sides of the y-tube have contrasting samples of "clean air" versus a "pheromone", that is supposed to attract the insect. We want to show that the pheromone is suitable to attract the insect. For the set-up of the experiment see video: https://youtu.be/6dwy7HcCuVI

To perform an exact binomial test in R, you can use the function *binom.test()* for your calculations. The function can take the following arguments:

```
binom.test(x, # number of successes
           n, # number of trials
           p = 0.5, # hypothesized probability of success
           alternative = c("two.sided", "less", "greater"),
           conf.level = 0.95)
```

Consult the help of this function (`?binom.test`) to read more details about the exact binomial test in R.

Write down the steps of the (exact) binomial test (with $\alpha = 0.05$) and give the conclusions in the following situations using the p-value method:

a. 40 insects choose the pheromone over the clean air sample,

b. 50 insects choose the pheromone over the clean air sample,

c. 60 insects choose the pheromone over the clean air sample,

d. 80 insects choose the pheromone over the clean air sample,

e. 100 insects choose the pheromone over the clean air sample.

f. How many silkworm moths are minimally required to have significant proof, that the "pheromone" attracts silkworm moths?

3.2.2 Exercise Binomial Test: Sleep apnea

Obstructive sleep apnea is a sleep disorder, that causes a person to stop breathing momentarily and then awaken briefly. Researchers of a university found that 7 of 20 commercial truck drivers suffered from obstructive sleep apnea. Sleep researchers theorize that 25% of the general population suffers from obstructive sleep apnea.

a. Investigate using the p-value method and $\alpha = .05$, whether the percentage of truck drivers suffering from obstructive sleep apnea is larger than 25%. Mention all the steps. You can use *binom.test()* for this exercise or otherwise familiarize yourself with the function *pbinom()* to calculate the appropriate probabilities yourself.

b. Investigate by mentioning all the steps when using method of the rejection region and $\alpha = .05$, whether the percentage of truck drivers suffering from obstructive sleep apnea is larger than 25%.

3.3 Command Reference

Table 3.1: Lesson 3 commands

Command	Description
array()	create a multi-way array
as.factor()	change object into a factor variable
as.matrix()	convert argument into a matrix
binom.test()	perform an exact Binomial test
cbind()	combine objects by columns
colnames()	retrieve or set column names of a matrix object
complete.cases()	find complete cases in an object (without `NA`)
data.frame()	create a data frame object
data.matrix()	convert a data frame to a numeric matrix
dim()	retrieve or set the dimension of an object
dimnames()	retrieve or set the dimnames of an object
factor()	encode a vector as a nominal categorical factor
head()	return the first part of an object
levels()	retrieve or set the levels attribute of an object
list()	construct a list object
match() or `%in%`	value matching, `%in%` returns logical vector
matrix()	create a matrix object
ncol()	retrieve the number of columns of an object
nrow()	retrieve the number of rows of an object
ordered()	encode a vector as an ordinal categorical factor
pbinom()	binomial probability distribution function
rbind()	combine objects by rows
read.table()	read file in table format, create data frame
rownames()	retrieve or set the row names of a matrix object
row.names()	get and set row names for data frames
str()	display compactly structure of an arbitrary object
subset()	return subsets of vectors, matrices or data frames which meet conditions
table()	cross tabulation and table creation
tail()	return the last part of an object
unclass()	strip object of class information
View()	invoke RStudio's data viewer
with()	evaluate a logical expression in a data environment

Lesson 4

Dynamic Reports, Correlation and Regression

4.1 Dynamic Reports in R

When doing statistical analyses, it would be nice to obtain a report directly from the software containing not only the results from the analysis, but also your own comments which you added to the analyses.

Now this is exactly what you can do within RStudio with the **knitr** package [7]. RStudio has a menu item, that you can click to obtain a report in either HTML, PDF or Word format. For this to work you should write not only the R code, but also your personal text into one single document. This document should have a specific format: it should have the `Rmd` format. `Rmd` stands for "R markdown". R markdown is a variant of markdown, which is a markup language to write documents in plain text format, which are easily converted into HTML or other formats. The markup adds semantics (meaning) to the syntax (your plain text).

The report produced in this way is **dynamic** [8], meaning that if anything changes in the data set or statistical analysis a new updated report is almost immediately obtained. It is also important in the sense that research becomes statistically reproducible by combining written text with the actual calculations [9].

4.1.1 Dynamic Reports: Rmd and knitr

In this lesson you get an introduction into R markdown, so that at the end of the day you are able to do statistical analyses, and add comments and conclusions to produce a concise report. You will work by example.

- Within Rstudio create a new Rmd file: Click in the top menu of RStudio **File** → **New File** → **R Markdown**...
- Fill in the Title, e.g. "My first R report", and the Author, e.g. "My name". Choose the Output format `PDF`.

Now, RStudio will make a `*.Rmd` file in the code editor, as shown in Figure 4.1, and fills it with some relevant, but also irrelevant content:

- First save this `*.Rmd` file, by clicking in the top menu of RStudio **File** → **Save As**... and give it a name you like, in the folder that you prefer.
- Next, click on the "Knit" button and the result will be your first R report, in pdf format, obtained after a few clicks only! The `Rmd` file is processed by functions in the **knitr** package, which render (or knit) your text and calculation code together. This package is meant for what is called **Dynamic report generation**. If you change anything in the data file or analysis (inside the `Rmd` file), a simple click on the "Knit" button will produce a new updated report as pdf document.

Let's go through the template `Rmd` file in detail:

- Locate the **code chunks**. These are the pieces of text that start with ``` (three back ticks) and end with ```. In these chunks you put (most of) the R code that you want to run. How many chunks do you see in the Rmd file?

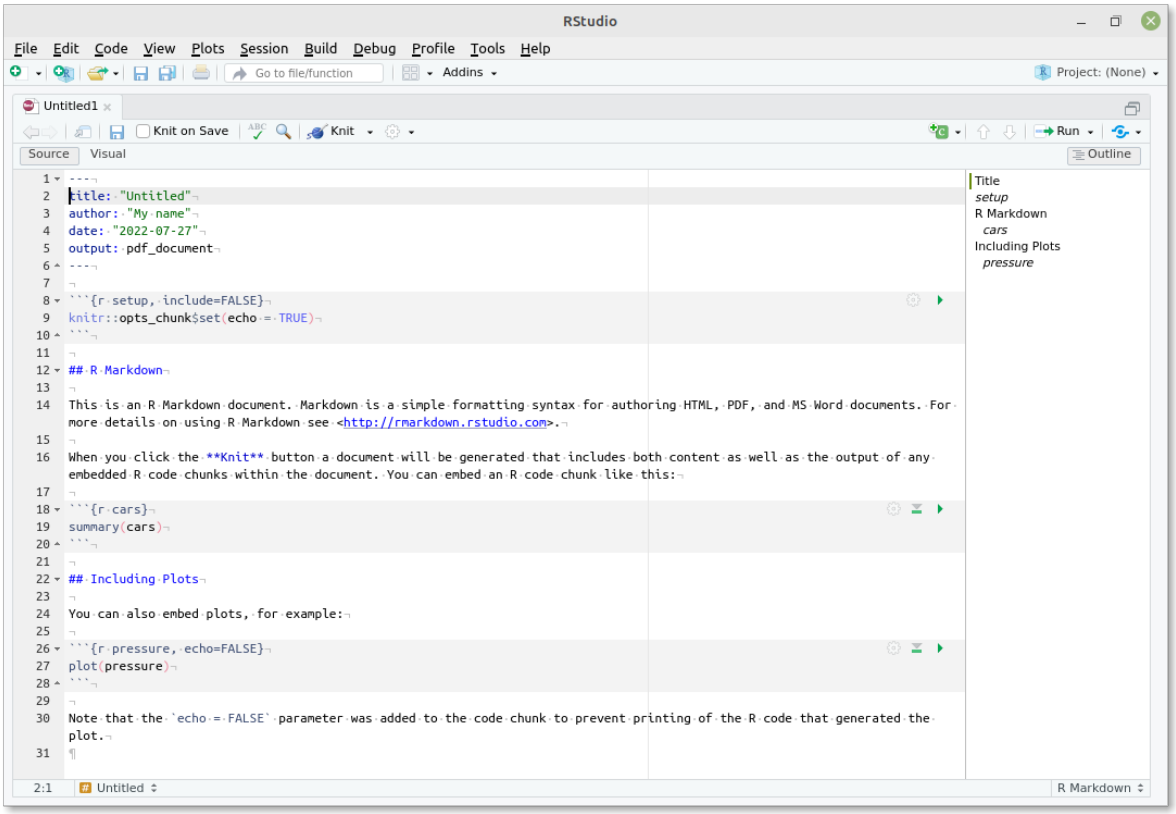


Figure 4.1: Screenshot of the R Markdown template.

  - A chunk starts with ```` ```{r chunk-label} ```` . You can choose the name chunk-label yourself. Make sure that different chunks have different chunk-labels!
  - After the chunk-label, various chunk options can be specified, which tell to **knitr** how to behave. In the current example we see the following:
    * ```` ```{r setup, include=FALSE} ````: `include=FALSE` tells that **knitr** will run the R code in the chunk, but neither the R commands nor the R output will be included in the report. This is useful for this chunk, because it is only meant to set up some default behavior
    * ```` ```{r pressure, echo=FALSE} ````: `echo=FALSE` tells that **knitr** will not display the R code in the chunk in the report.
    * ```` ```{r cars} ````: no option is specified. Now, by default, **knitr** will include both the R code and the R output into the report.
  - Useful other chunk options are:
    * `eval=FALSE`: **knitr** will not run the R code in the chunk, so no R output can and will be shown in the report.
    * `results='hide'`: now **knitr** will run the R code, but no results are shown.
    * `warning=FALSE`: **knitr** will not display any warning message generated by the R code.
    * `fig.keep='none'`: **knitr** will not keep any figure produced in the chunk, therefore the figure will not be displayed.
  - There are many more chunk options. Have a look at the help for R Markdown and Markdown within RStudio: click in the top menu of RStudio on **Help** → **Markdown Quick Reference**, or on **Help** → **Cheatsheets** → **R Markdown Cheat Sheet**/**R Markdown Reference Guide**. The last two Cheat Sheets require internet connection and will download a file or open in a browser window.

- Locate the pieces of texts between the chunks. Note the following:
  - The `Rmd` file starts with the specification of title, author name and date, as you would normally do in a report. This is done here using the reserved words `title`, `author`, `date`, between lines containing `---`.

- A heading is a line of text preceded by one or more `#` symbols, like `## R Markdown`. If you use more `#` symbols, smaller fonts will be used for the heading.
- You can separate paragraphs by an empty line.
- Bold face **text** is produced by `**text**` or `__text__`.
- Useful other Markdown syntax:
  * `_text_` or `*text*` produces italic *text*
  * `x^2^` produces superscript x^2
  * `x~2~` produces subscript x_2
  * a mathematical formula, e.g. regression equation $y = \alpha + \beta x + \varepsilon$, is produced by `$y = \alpha + \beta x + \varepsilon$`.

There are many more options. Again, have a look at the help for R Markdown and Markdown within RStudio.

4.1.2 Exercise: Dynamic Reports

We return to the R script about the binomial distribution, which was discussed yesterday. Read this R script, which is available on Brightspace as file `R4S_day3.R`, into RStudio: click in the top menu of RStudio on **File** → **Open File**...; look for file `R4S_day3.R`.

Make a report based on the statistical analysis as shown in this R script. You may take the following steps:

- Copy and paste the content of file `R4S_day3.R` into the Rmd file you created in Lesson 3 or downloaded from Brightspace. Save this file e.g. under the name: `R4S_day3.Rmd`.
- Add a header to the report.
- Split the content of the R script in relevant parts, and make sure that each part becomes a R chunk.
- Add your own comments and conclusions to each part. Do this in text and not all inside the code chunks!
- "Knit" the Rmd file into a report.

4.2 Correlation and Simple Linear Regression

In the second part of this lesson we study how R can be used for correlation and simple linear regression.

For theory see O&L Sections 11.1, 11.2, 11.3, 11.4, 11.6 [5] (6^{th} Edition [4]: Section 11.7 instead of 11.6)

4.2.1 Pearson Correlation Coefficient

If we have a pair of continuous variables x and y we may be interested in the strength of the linear relationship between the two. The strength of the linear relationship can be quantified with the Pearson correlation coefficient ρ. Based on a random sample from the population, we can estimate ρ with the sample correlation coefficient r:

$$r = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2 \sum_{i=1}^{n} (y_i - \bar{y})^2}} \tag{4.1}$$

The correlation coefficient can also be written as

$$r = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{s_x \times s_y} \frac{1}{n-1} = \sum_{i=1}^{n} \frac{(x_i - \bar{x})}{s_x} \frac{(y_i - \bar{y})}{s_y} / (n-1) = \sum_{i=1}^{n} \frac{z_{x_i} \times z_{y_i}}{n-1} \tag{4.2}$$

, with z_{x_i} and z_{y_i} being the standardized versions of variables x_i and y_i. In the formula above s_x and s_y represent the sample standard deviations of x and y respectively.

Below we take a very simple example. Imagine that there are 5 males, randomly drawn from some population, on which length x (in meters) and weight y (in kg) are measured. We want to estimate the correlation coefficient ρ.

```
x <- c(1.71, 1.84, 1.78, 1.91, 1.68)
y <- c(68, 81, 72, 88, 70)
cor(x, y)
```

```
#> [1] 0.9534577
```

You can see that the correlation is 0.95, which indicates a strong positive linear relationship between length and weight.

Variables can be standardized (which means subtracting the mean value and dividing by the standard deviation) by applying the R function *scale()*. Below you can see that the same value of r is obtained using Equation 4.2 as with the *cor()* function.

```
zx <- scale(x)
zy <- scale(y)
n <- length(x)
(r <- sum(zx * zy) / (n - 1))
```

```
#> [1] 0.9534577
```

Testing H_0: $\rho = 0$ versus H_a: $\rho \neq 0$ can be done with the R function *cor.test()*:

```
cor.test(x, y)
```

```
#>
#>  Pearson's product-moment correlation
#>
#> data:  x and y
#> t = 5.4769, df = 3, p-value = 0.01197
#> alternative hypothesis: true correlation is not equal to 0
#> 95 percent confidence interval:
#>  0.4483186 0.9970239
#> sample estimates:
#>       cor
#> 0.9534577
```

The resulting t-test statistic is equal to 5.477 with a p-value $0.012 \leq 0.05$ (where 0.05 is the chosen significance level α). Therefore, H_0 is rejected.

4.2.2 Exercise: Correlation

Read the file `grasshopper.csv` into R using *read.csv()* and show the first few observations using function *head()*. Variable x is the weight of the grasshopper in grams and y is the number of eggs laid by the grasshopper. We believe that there is a linear relationship between weight of the grasshopper and number of eggs and want to assess the strength of this relationship. Calculate the correlation coefficient r of x and y in two ways:

1. using the *cor()* function.
2. using a formulate for r based upon standardized x and y.

Test H_0: $\rho \leq 0$ against H_a: $\rho > 0$. Figure out yourself how the right-sided alternative hypothesis should be specified, by looking into the help for *cor.test()*. What do you conclude about the correlation?

Make a scatterplot of y versus x and comment on the usefulness of the correlation to quantify the relationship between x and y.

4.2.3 Simple Linear Regression

The simple linear regression model describes a linear relationship between dependent variable y and regressor (explanatory variable) x:

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i \tag{4.3}$$

, with $i = 1, \ldots, n$ an index for observation number. We assume that the errors ε_i are independent, have constant variance and are normally distributed with expected value 0.

Least-squares estimates of the parameters β_0 and β_1 are obtained using the R function *lm()*. This function wants a model formula `y ~ x` as input. Below you see that the result from the linear regression is saved into the R object ***lmo*** (of class *lm*). The object name, here given as ***lmo***, you can, of course, choose yourself in the assignment.

Below we look at an example about quality of fish (y) after delayed storage on ice (x). The delay was either 0, 3, 6, 9, or 12 hours. The quality of fish is a score between 0 and 10. Please type the commands shown below into a R script, or, even better, into a Rmd file.

```
x <- c(0, 0, 3, 3, 6, 6, 9, 9, 12, 12)
y <- c(8.3, 8.5, 7.3, 7.5, 7, 6.5, 6.2, 5.9, 5.9, 5.7)
lmo <- lm(y ~ x)
class(lmo)
```

```
#> [1] "lm"
```

To obtain results from the regression we have to manipulate the ***lmo*** object further. Let's check out different possibilities:

- Plotting the regression line into a scatterplot: R function *abline()*

```
# add more narrow space around plot
par(mar = c(2.5, 2.5, 2, 1),
    mgp = c(1.5, 0.35, 0),
    cex = 0.8)
# create the scatterplot with names on the axes and a figure title
plot(y ~ x,
     xlab = "delay",
     ylab = "fish quality",
     main = "Regression fish quality")
# add the regression line
abline(lmo, lty = "dashed", col = "red")
```

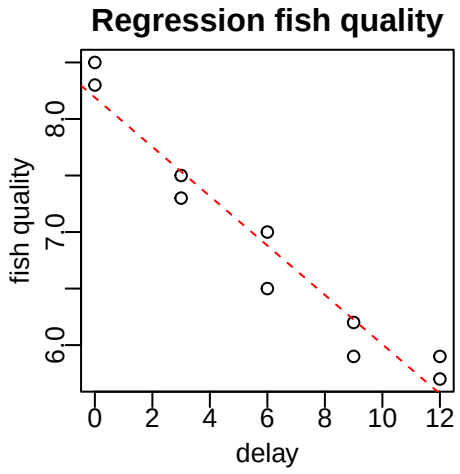


Figure 4.2: Scatterplot with fitted simple linear regression line for the Fish Quality data

- Extracting the least-squares estimates for intercept and slope: R function *coef()*}

```
coef(object = lmo)
```

```
#> (Intercept)           x
#>   8.1900000  -0.2183333
```

- Summarizing the linear regression: R function *summary()*. Many quantities are produced, like the least-squares estimates of the regression coefficients with standard errors, t-tests and P-values; residual

standard deviation; R^2 and R^2_{adj}; omnibus F-test statistic with p-value, testing whether all coefficients (except the intercept) are zero.

```
summary(object = lmo)
```

```
#>
#> Call:
#> lm(formula = y ~ x)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -0.3800 -0.1850  0.0425  0.1275  0.3300
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  8.19000    0.14433   56.74 1.03e-11 ***
#> x           -0.21833    0.01964  -11.12 3.83e-06 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.2635 on 8 degrees of freedom
#> Multiple R-squared:  0.9392, Adjusted R-squared:  0.9316
#> F-statistic: 123.6 on 1 and 8 DF,  p-value: 3.832e-06
```

- The function *coef()* can also be applied to the output of the *summary()* function. The result is a matrix, containing the estimates, standard errors, t-statistics and P-values in the columns. Elements from this matrix can be subsetted in the usual way:

```
(coefmat <- coef(summary(object = lmo)))
```

```
#>               Estimate Std. Error   t value     Pr(>|t|)
#> (Intercept)  8.1900000 0.14433035  56.74482 1.032591e-11
#> x           -0.2183333 0.01964087 -11.11627 3.831914e-06
```

```
coefmat[2, 1:3]
```

```
#>     Estimate   Std. Error      t value
#>  -0.21833333   0.01964087 -11.11627443
```

- Confidence interval for the regression coefficients are obtained with the *confint()* function:

```
confint(object = lmo, level = 0.99)
```

```
#>                 0.5 %     99.5 %
#> (Intercept)  7.7057158  8.6742842
#> x           -0.2842361 -0.1524306
```

- Extract the residuals and predicted (fitted) values: R functions *residuals()* and *fitted()*:

```
res <- residuals(object = lmo)
pred <- fitted(object = lmo)
```

- Extract the residual sum of squares with the R function *deviance()*:

```
deviance(object = lmo)
```

```
#> [1] 0.5555
```

- Extract the residual degrees of freedom with the R function *df.residual()*. In simple linear regression this is $n - 2$ (can you explain why?):

```
df.residual(object = lmo)
```

```
#> [1] 8
```

- Extract the estimate of the residual, or error, standard deviation $\hat{\sigma}_\varepsilon$. This value is part of the *summary()* output, which is a list. The error standard deviation is a component with name `sigma` from this list:

```
sigma(lmo) # or equivalently: summary(object = lmo)$sigma
```

```
#> [1] 0.26351
```

- Assumptions need to be checked with residual plots. Typical plots are a normal Q-Q plot of the residuals (to check normality) and the plot of residuals versus fitted (predicted) values (to check whether the assumed model is the best fitting model and constant variance).

```
# two plots next to each other
par(mfrow = c(1, 2))
# Normal Q-Q Plot
qqnorm(res)
qqline(res, col = "red")
# scatterplot of residuals vs. predicted values
plot(res ~ pred)
# add horizontal line res = 0
abline(h = 0)
title("Homoscedasticity check")
```

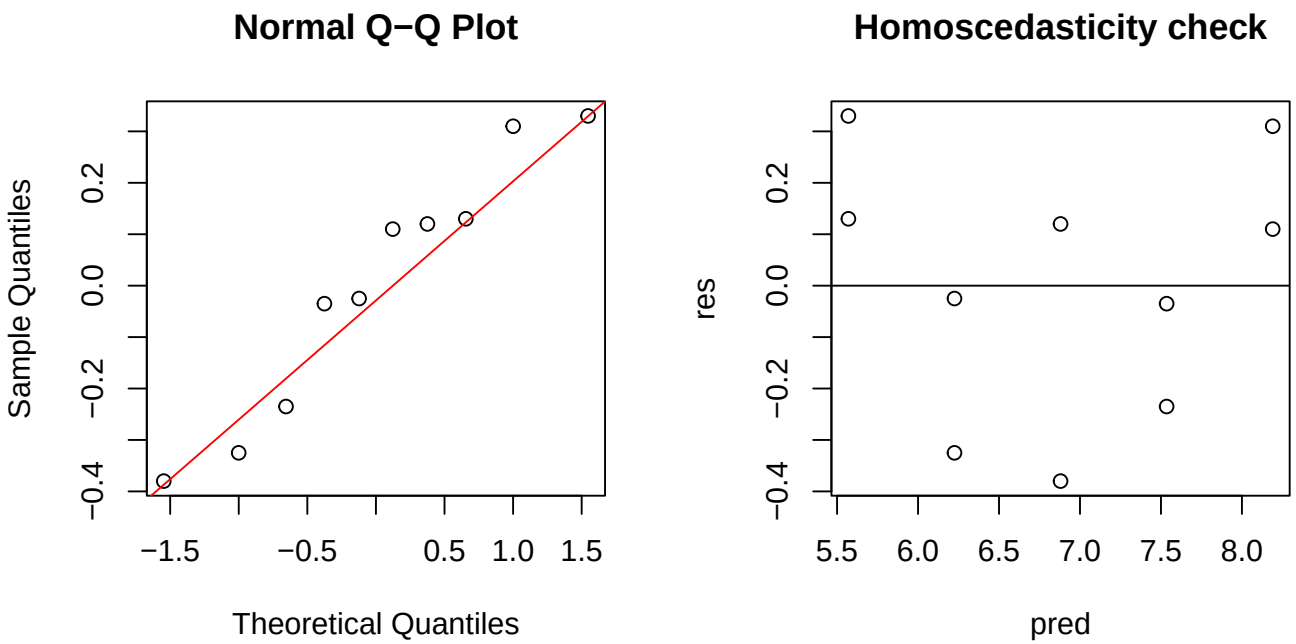


Figure 4.3: Plots to check the assumptions for simple linear regression

- Estimation of the mean response with standard errors and confidence intervals at given new values of x can be obtained with the *predict()* function. First a data frame containing the new data must be defined, where the name of the explanatory variable (predictor) **must** match the one used in the model (in this case the predictor was named `x`).

```
newdata <- data.frame(x = c(5.5, 6.5))
predict(object = lmo,
        newdata = newdata,
        se.fit = TRUE,
        interval = "confidence")
```

```
#> $fit
#>        fit      lwr      upr
#> 1 6.989167 6.795679 7.182654
#> 2 6.770833 6.577346 6.964321
#>
#> $se.fit
```

```
#>          1          2
#> 0.08390585 0.08390585
#>
#> $df
#> [1] 8
#>
#> $residual.scale
#> [1] 0.26351
```

- The same function can be used to get predictions of future (new) observations with prediction intervals at given new values of the explanatory variable (predictor) (x in this case).

```
newdata <- data.frame(x = c(5.5, 6.5))
predict(object = lmo,
        newdata = newdata,
        interval = "prediction")
```

```
#>        fit      lwr      upr
#> 1 6.989167 6.351450 7.626883
#> 2 6.770833 6.133117 7.408550
```

> ⚠ Warning: Name predictor in data.frame for *predict()*
>
> Make sure that in the ***newdata*** object the variable specified for the explanatory variable matches exactly the name of the explanatory variable in the object containing the linear model. In the example above the explanatory variable in the ***lmo*** object is specified as x, which is the same as in the ***newdata*** object.

4.2.4 Exercise: Simple Linear Regression

Return to the data on grasshoppers. Make a small report on the statistical analysis of the dataset, showing the R code and R output, and answering the questions:

a. Read the data file `grasshopper.csv` into R (again) and show the content in a sensible way (check the Tip: Looking at the data in Section 3.1.7).

b. Fit a simple linear regression model, explaining the number of eggs y from the weight of female grasshopper x.

c. Plot y versus x and draw the regression line.

d. Summarize the regression output using the *summary()* function.

e. Obtain the slope for weight and give its interpretation.

f. Test H_0: $\beta_1 \leq 0$ versus H_a: $\beta_1 > 0$. Why does it make sense to test whether the slope is positive? Mention the outcome of the test statistic, the p-value and the conclusion.

g. Write down the fitted regression model.

h. Give R^2.

i. Estimate the mean number of eggs for a grasshopper with weight of 3 grams, and give a 95% confidence interval for it.

j. Check the assumptions of normality and constant variance.

k. Make a plot of residuals versus regressor x. What do you see?

l. Comment on the model fit.

4.3 Command Reference

Table 4.1: Lesson 4 commands

Command	Description
abline()	add straight lines to a plot
coef()	extract coefficient from a fitted model object
confint()	compute confidence intervals for parameters in a fitted model object
cor()	compute the correlation between two vector objects
cor.test()	test for association/correlation between paired samples
deviance()	returns the deviance of a fitted model object
df.residuals()	returns the residual degrees-of-freedom from a fitted model object
fitted()	extract fitted model values from a model object
lm()	function for fitting a linear model
par()	set or query graphical parameters
predict()	generic function for predictions with a fitted model object
plot()	generic function to create a plot, a.o. $\boldsymbol{y}$ vs. $\boldsymbol{x}$ plot (scatterplot)
qqline()	adds a line to a "theoretical", by default normal, quantile-quantile plot
qqnorm()	generic function to create a normal Q-Q plot
residuals()	generic function to extract residuals from a fitted model object
scale()	generic function to center and scale the columns of a numeric matrix object
sigma()	extract the residual, or error, standard deviation from a model object
sum()	calculate the sum of vector elements
title()	plot annotation function to add labels to a plot

Lesson 5

End of Week 1 Assignment

The Netherlands National Institute for Public Health and the Environment (RIVM) researches many topics, among others food safety. A couple of years ago RIVM was asked to make a risk assessment for the public health with respect to bacteria in spices sold in Asian supermarkets (Tokos) throughout the Netherlands.

In this assignment you will analyze a part of this data, which were collected for the risk assessment calculation. The data concern *Bacillus cereus* (abbreviation: *B. cereus*) concentrations reported in Colony Forming Units (CFU) per gram in the spices:

1. cardamom (*Elettaria cardamomum*), a.k.a. green or true cardamom, and
2. turmeric (*Curcuma longa*).

Background information on the bacterium and the spices in the RIVM research can be found on these websites (or find your own sources):

B. cereus https://www.foodsafety.gov/poisoning/causes/bacteriaviruses/bcereus/
cardamom https://en.wikipedia.org/wiki/Cardamom
turmeric https://en.wikipedia.org/wiki/Turmeric

Write a report in R markdown using the template from Brightspace, where you answer the questions given below. All data, required to answer the questions below, is contained in a `RData`—file named `week1_assignment.RData`, which you can find in your course data directory. You can load the data with the following command:

```
load("./week1_assignment.RData")
```

You will have to adapt the path to where you have stored the course data files, to be able to load the data yourself.

When you have loaded the week 1 assignment data, you will find two objects in the workspace environment (***data1***, and ***data2***). The ***data1*** object contains the data needed for answering the questions a.-f., and consists of two simple random samples drawn from jars of cardamom and turmeric obtained from Tokos throughout the Netherlands.

Submit your report as a rendered PDF-file before the deadline, otherwise it will not be taken into account and you will not receive feedback on your work.

a. Make graphical representations for the *B. cereus* concentrations for cardamom and turmeric. Is the assumption reasonable that the concentrations are drawings from normal distributions?

b. Transform the data by calculating the 10 base logarithm of the *B. cereus* concentration (check Table 1.2). [Hint: create a data frame with a third column named `log10conc` and use the correct function!] Check the assumption of normality based on the transformed data by making sensible plots and tests. Explain what you can conclude from these plots and tests.

c. Calculate the mean, standard deviation for the ^{10}log *B. cereus* concentration of both cardamom and turmeric. Construct the 95% and 99% confidence interval for the mean ^{10}log *B. cereus* concentration of both spices.

d. Is the 95% confidence interval smaller or wider than the 99% confidence interval? Explain why!

e. Investigate whether there is a difference between the mean ^{10}log *B. cereus* concentration in cardamom and turmeric. Use a significance level of 0.05 and mention all steps you use to draw your conclusion (Do not forget to give the conclusion in words for readers lacking a statistical background). For any statistical test the minimal required steps are:

1. the appropriate parameter(s) definition(s) in words,
2. the definition of the null hypothesis H_0 and alternative hypothesis H_a,
3. the distribution of the test statistic under H_0,
4. the outcome of the test statistic and the p-value, or a properly documented p-value,
5. the conclusion based on the p-value and the significance level (α), as well as the conclusion in words.

Assume that extensive research has shown that the mean ^{10}log *B. cereus* concentration in cardamom sold in supermarkets equals 2.9.

f. Investigate whether the expected ^{10}log *B. cereus* concentration in cardamom sold in Tokos differs from the **mean** ^{10}log B. cereus concentration in cardamom sold in supermarkets (use $\alpha = 0.01$). Mention all steps to reach the conclusion (also in words!).

RIVM suspects, that storage conditions (mainly the storage temperature) during transport and in stores are an important reason for the high concentrations B. cereus in the spices. To investigate this suspicion the following (fictional) experiment was performed: a total of 12 randomly chosen jars of cardamom were randomly assigned to 4 different temperatures (3 jars per temperature, with temperatures being: 15, 20, 25 and 30°C). After 3 months of storage under the assigned temperature the *B. cereus* concentration was determined in CFU/gram and the ^{10}log *B. cereus* concentration was calculated. The data for this (fictional) experiment are contained in the ***data2*** object in your workspace environment. Use the ***data2*** to answer the questions g. to m. in your R markdown report.

g. Make a meaningful graphical representation of the data contained in ***data2*** by using the *plot()* function and additionally needed functions.

h. Give the equation for the relevant least squares regression line and remember to give meaning to your variables.

i. Give the interpretation, using ^{10}log concentration *B. cereus* in your description, for the coefficients of the least squares regression line.

j. Test whether the mean ^{10}log *B. cereus* concentration increases, when the storage temperature increases (use $\alpha = 0.05$). Mention all steps needed to reach your conclusion in words for a person with no statistical background.

k. Give the rejection region, for the test you performed in the previous question.

l. Provide an estimate for the mean ^{10}log *B. cereus* concentration at a temperature of 27°C.

m. Provide a 95% confidence interval for the mean ^{10}log *B. cereus* concentration at a temperature of 27°C.

n. Provide a 95% prediction interval for the ^{10}log B. cereus concentration for a new observation in the future at a temperature of 27°C.

o. Is the prediction interval smaller or wider than the confidence interval? Explain your answer.

5.1 Command Reference

Table 5.1: Lesson 5 commands

Command	Description
load()	reload data sets (a `*.RData`—file) written with the function *save()*
save()	write an external representation of R objects to a specified `*.RData`—file

Part II

Week 2

Lesson 6

R Coding Style, Packages, Control Structures, ANOVA and Pairwise Comparison

6.1 R coding style, Packages and Control Structures

6.1.1 R coding style

When writing scripts and later on functions, good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read. As with styles of punctuation, there are many possible variations. The following guide describes the style, which is commonly used (this reader tries to follow this style as closely as possible). It is based on Google's R style guide (https://google.github.io/styleguide/Rguide.xml), with a few tweaks here and there.

You don't have to use this style, but it is strongly recommended that you use a consistent style and you document it. If you're working on the code of someone else, do not impose your own style. Instead, read their style documentation and follow it as closely as possible.

Good style is important because while your code only has one author, it will usually have multiple readers. This is especially true when you are writing code with others. In that case, it is a good idea to agree on a common style up-front. Since no style is strictly better than another, working with others may mean that you will need to sacrifice some preferred aspects of your style.

The **formatR** package, by Yihui Xie [10], makes it easier to clean up poorly formatted code. It can not do everything, but it can quickly get your code from terrible to pretty good. Make sure to read the notes on the website (http://yihui.name/formatR/) before using it. It is as easy as:

```
install.packages("formatR")
formatR::tidy_dir("R")
```

A complementary approach is to use a code **linter**. Rather than automatically fixing problems, a linter just warns you about them. The **lintr** package by Jim Hester [11] checks for compliance with this style guide and lets you know, where you have missed something. Compared to **formatR** it picks up more potential problems, because it does not need to fix them also. However, you will still see false positives.

```
install.packages("lintr")
lintr::lint_package()
```

6.1.1.1 Object Names

Variable and function names should be lowercase. Use an underscore ("_") to separate words within a name (reserve "." for S3 methods, which will not be discussed in this reader). Camel case (see https://en.wikipedia.org/wiki/Camel_case) is a legitimate alternative, but be consistent! Generally, variable names

should be nouns and function names should be verbs. Strive for names that are concise and meaningful (this is usually not as easy as it sounds!).

```
# Good
day_one
day_1
dayOne # example of lower camel case
DayOne # example of upper camel case

# Bad
first_day_of_the_month
dayone
djm1
```

Whenever possible avoid using names of existing functions and variables. The use of existing names can potentially greatly confuse the readers of your code.

```
# Bad
T <- FALSE
c <- 100
mean <- function(x) sum(x)
```

6.1.1.2 Spacing

Place spaces around all infix operators ("=", "+", "-", "<-", *etc.*). The same rule applies when using "=" in function calls. Always put a space after a comma, and never before (just like writing English text in general).

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

There is a small exception to this rule: ":", "::" and ":::" do not need spaces around them. (You have not seen "::" or ":::" before, do not wreck your brain over this - you'll learn about them in this lesson, where packages are discussed.)

```
# Good
x <- 1:10
base::get

# Bad:
x <- 1 : 10
base :: get
```

Place a space before the left parentheses, **except** in a function call.

```
# Good
if (debug) do(x)
plot(x, y)

# Bad
if(debug)do(x)
plot (x, y)
```

Extra spacing (i.e., more than one space in a row) is allowed if it improves alignment of equal signs or assignments ("<-"), thereby enhancing readability of your code.

```
list(
  total = a + b + c,
  mean  = (a + b + c) / n
)
```

Do not place spaces around code in parentheses or square brackets (unless there is a comma, in which case use the coding style about spaces as stated above).

```
# Good
if (debug) do(x)
diamonds[5, ]

# Bad
if ( debug ) do(x)  # No spaces around debug
x[1,]   # Needs a space after the comma
x[1 ,]  # Space goes after comma not before
```

6.1.1.3 Curly braces

An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line, unless it is followed by `else`.

Always indent the code inside curly braces.

```
# Good
if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}

# Bad
if (y < 0 && debug)
message("Y is negative")

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

It is all right to leave very short statements on the same line:

```
if (y < 0 && debug) message("Y is negative")
```

6.1.1.4 Line length

Strive to limit your code to 80 characters per line. This fits comfortably on a printed page with a reasonably sized font. If you find yourself running out of room, this is a good indication that you should encapsulate some work in a separate function.

6.1.1.5 Indentation

When indenting your code, use two spaces. Never use tabs or mix tabs and spaces. Change these options in the code preferences pane, as shown in Figure Figure 6.1.

The only exception is, if a function definition runs over multiple lines. In that case, indent the second line to where the definition starts:

```
long_function_name <- function(a = "a long argument",
                               b = "another argument",
                               c = "another long argument") {
```

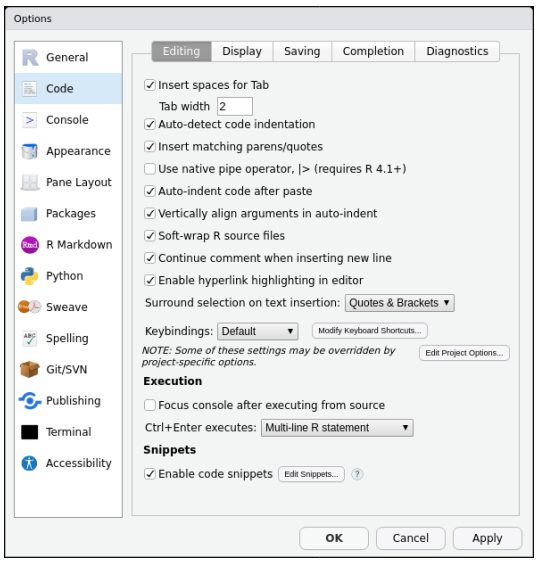


Figure 6.1: Coding style options preference pane from the RStudio top menu Tools → Gobal Options...

```
  # As usual code is indented by two spaces.
}
```

6.1.1.6 Assignment

Use "<-", not "=", for assignment as explained in Section 1.4.3. The "=" is reserved for specifying default values in a function definition.

```
# Good
x <- 5

# Bad
x = 5
```

6.1.1.7 Commenting guidelines

Comment your code. Each line of a comment should begin with the comment symbol and a single space: "#". Comments should explain the why, not the what.

Use commented lines of "-" and "=" to break up your file into easily readable chunks. The minimal amount of "-" or "=" is **four** without spaces in between. This will enable you to fold and unfold your code in the code editor of RStudio.

```
# For example comment parts of your code like:
# Load data ---------------------------
# Plot data ===========================
```

6.1.2 Packages

Packages are collections of R functions, data, and compiled code in a well-defined format. When you install a package, it gives you access to a set of commands that are not available in the **base** R set of functions. The directory, where packages are stored, is called the library. R comes with a standard set of packages. Others are available for download and installation. Once installed, a package has to be loaded into the session to be able to use the functions and data contained in that specific package.

To see which packages are already installed and available to you in R, execute the *library()* function without arguments or use the *installed.packages()* function. In RStudio you can also click on the Packages tab in the "Plots and Files" section, panel 4 in Figure 1.5. There all available packages are listed alphabetically by name, with description and version.

6.1.2.1 Installing

If a certain package, you would like to use, is not yet available, you will need to install it. By installing a package you extend the functionality of R with new functions. In Section 1.6.1 you have already done this for the **swirl** package via a console command:

```
install.packages("swirl")
```

In the same section you can also read, how to install packages using the graphical interface of RStudio.

By default, packages are installed from the Comprehensive R Archive Network (https://cran.r-project.org/), commonly known as **CRAN**, or one of its mirrors. As stated in lesson 1, nowadays, there are over 18700 packages on CRAN you can use. To find the package you need, requires searching and reading on the internet. In other words, as quoted before "***Google is your friend***".

To make things worse there is another repository, with loads and loads, of packages. This repository was also already mentioned in lesson 1 and is called the Bioconductor project (https://bioconductor.org) and contains open source software for bio-informatics. Some packages on CRAN might even be dependent on packages available on Bioconductor. So be aware when installation of a CRAN package fails, because it can not find a specific dependency. The dependency might be located on Bioconductor!

6.1.2.2 Loading

Only when a package is loaded, are its contents (functions and included data sets) available within R. This is done both for efficiency (the full list would take more memory and would take longer to search than a subset), and to aid package developers, who are protected from name clashes with other code (functions or data sets in other packages).

To load a particular package, e.g. the **boot** package [12,13] issue the following command:

```
library(boot)
```

When using the *library()* function for loading a package, the package name may be between quotes (single or double). However, this is not a necessity. In the console of RStudio the auto-completion functionality will cause the package name to be unquoted, when using the *library()* function to load a desired package. In RStudio you can also tick the box in front of a package name on the Packages tab in the "Plots and Files" section, panel 4 in Figure 1.5. When loading a package using the graphical interface of RStudio, the command used in the background gets printed in the console.

When a package is loaded, it may contain a function that has the same name as a function from another package. In this case lexical scoping (like searching a dictionary or a name list) in R becomes important. The package order, in which packages are loaded, determines, from which package the function is called. So if, say **packageX** and **packageY** both contain a function named *functionZ()*, the order in which both packages are loaded determines from which package the function is called. When **packageY** is loaded after **packageX**, the function *functionZ()* will be called from **packageY**, and vice versa. You can think of it in this way, the function name search list is extended on the front side of the search list, when you load a package. Therefore, R will encounter the function *functionZ()* for the first time in **packageY**, when it is loaded later than the **packageX**.

To explicitly call a function from a certain package you can use the double colon operator "`::`". For example if there is a function *get()* defined in another package, for example named **brapirv2**, and you want to use that specific *get()* function instead of the one in the **base** package you can call it like this:

```
brapirv2::get("object")
```

There is also a triple colon operator "`:::`". It is mainly used by R package developers and concerns function calls for functions hidden from the Namespace. Do not worry about this statement, as it is beyond the scope of "R for Statistics" and will not be discussed any further.

The double colon "`::`" operator has a very nifty property. The package from which the function is called, using the double colon operator, does not have to be loaded. By specifying both the package name and the function name R knows precisely (or uniquely as you will) which function to use. It also helps enormously, when you are writing functions and scripts yourself, to see from which package a certain function is coming from, that you are using. It aids the readability of your script or function.

There is another function to load packages in R. This is the *require()* function. Both, *library()* and *require()*, load the namespace of a package (available functions and data sets in this package) and attach it to the search list in R. The *require()* function is designed for use inside other functions, for example those you will develop yourself. It returns `FALSE` and gives a warning (rather than an error as *library()* does by default) if the package does not exist.

6.1.2.3 Unloading

When you can extend the functionality of R with loading packages, then there must also be a way to unload packages, thereby, decreasing the functionality or releasing identically named functions from the search list. To unload a previously loaded package you can execute the following command in the console:

```
detach("package:packagename", unload = TRUE)
```

where you replace **packagename** with the name of the package, you want to unload. See what happens when the **datasets** package is unloaded and you try to print the first six rows of the ***mtcars*** object:

```
# Unload datasets package
detach("package:datasets", unload = TRUE)
# Console print first 6 rows of mtcars
head(mtcars)
```

The above code should display the following message in the console: `Error in head(mtcars) : object 'mtcars' not found`.

Another way to unload a, previously loaded, package is by removing the tick in the box in front of the package name on the Packages tab in the "Plots and Files" section. Similar to loading a package using the graphical interface of RStudio, when unloading a package via RStudio the command, used in the background, is also printed in the console.

6.1.2.4 Updating

Sometimes packages are updated by the users, who created them. Updating packages can sometimes make changes to both the package and also to how your code runs. **If you already have a lot of code using a package, be cautious about updating packages as some functionality may change or disappear.**

Otherwise, go ahead and update old packages so things are up-to-date.

The following commands can be used:

```
# list all packages where an update is available
old.packages()

# update all available packages
update.packages()

# update, without prompts for permission/clarification
update.packages(ask = FALSE)

# update only a specific package, e.g. plotly, use install.packages()
install.packages("plotly")
```

The function *update.packages()* will update all packages in the known libraries interactively. This can take a while if you have not done it recently! To update everything without any user intervention, use the `ask = FALSE` argument.

In RStudio, you can also manage packages using **Tools** → **Install Packages**... from the top menu. The RStudio top menu option **Tools** → **Check for Package Updates**... I guess is quite self-explanatory. You can also use the buttons "Install" and "Update" at the top of the Packages tab in the "Plots and Files" section in RStudio.

6.1.3 Control structures

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are:

- `if...else` : testing a condition
- *switch()* : testing a condition and selecting one of a list of alternatives
- `for` : execute a loop a fixed number of times
- `while` : execute a loop while a condition is true
- `break` : break the execution of a loop
- `next` : skip an iteration of a loop
- `repeat` : execute an infinite loop

Most control structures are not used in interactive sessions (executing commands in the console), but rather when writing scripts, functions or longer expressions. Keep in mind that it is more efficient to use built-in functions rather than control structures, whenever possible.

6.1.3.1 `if...else` statement

Decision making is an important part of programming. One of the ways to achieve this in R programming is by using the conditional `if... else` statement. Figure 6.2 displays a flowchart for the `if...else` statement.

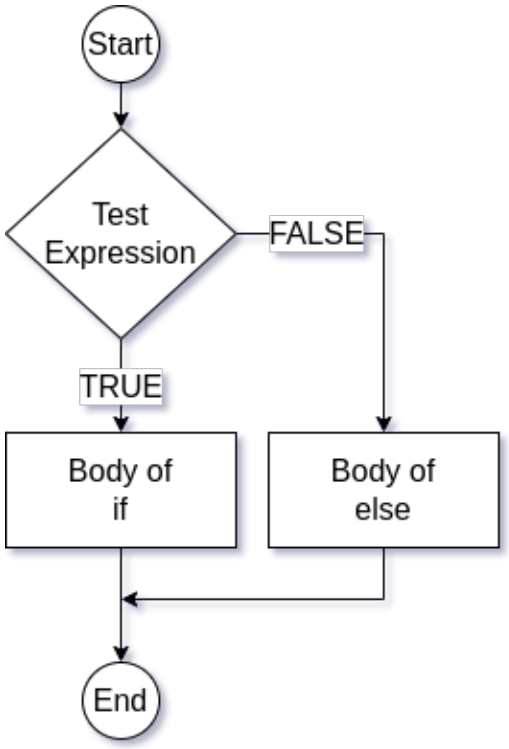


Figure 6.2: `if...else` statement flowchart

For example you could conditionally choose between two options with:

```
if (<condition>) {
  ## do something when <condition> == TRUE
  statement
  statement
...
} else {
  ## do something else when <condition> == FALSE
  statement
```

```
  statement
  ...
}
```

Notice that there is no semicolon behind each statement. When executing multiple commands in the console at once behind the prompt, you need to separate each command by a semicolon. When scripting or writing a function and using multiple statements in a control structure, this is not needed.

There is an easier way to use the `if...else` statement with respect to vectors in R programming. You can use the *ifelse()* function instead; the vector equivalent form of the `if...else` statement, e.g.:

```
x <- c(6:-4)
sqrt(x)
```

```
#>  [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000      NaN
#>  [9]      NaN      NaN      NaN
```

```
## Warning in sqrt(x): NaNs produced

sqrt(ifelse(x >= 0, x, NA))  # no warning
```

```
#>  [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000       NA
#>  [9]       NA       NA       NA
```

```
# even or odd
a <- c(5, 7, 2, 9)
ifelse(a %% 2 == 0, "even", "odd") # %% indicates the modulo operation, check ?`%%` for help
```

```
#> [1] "odd"  "odd"  "even" "odd"
```

The number of options given certain conditions, can be extended by merging multiple `if...else` statements, e.g.:

```
if (<condition1>) {
  ## do something when <condition1> == TRUE
  statement
  statement
  ...
} else if (<condition2>) {
  ## do something different when <condition1> == FALSE but <condition2> == TRUE
  statement
  statement
  ...
} else {
  ## do something different when both <condition1> and <condition2> are FALSE
  statement
  statement
  ...
}
```

The `if...else` statement is also a valid, when used without the `else` clause. In Figure 6.2 this would mean, that the body of `else` is none existing. So the following is perfectly legitimate:

```
if(<condition1>) {
  statement
  ...
}
# another conditional if statement
if(<condition2>) {
  statement
  ...
}
```

Assignment to store results in an object can both be performed inside and outside an `if...else` statement. The following two examples are **both valid** `if...$else` control structures:

```
# assignment inside the if...else control structure
if (x > 3) {
  y <- 10
} else {
  y <- 0
}

# assignment outside the if...else control structure
y <- if (x > 3) {
  10
} else {
  0
}
```

6.1.3.2 *switch()* function

The *switch()* function is very much related to the `if...else` statement. Instead of programming multiple nested `if...else` statements to select an option, you can also use *switch()*. This function lets you select an option of a list of alternatives after evaluation of an expression, that should result in a character string or a number. Based on the outcome an option is chosen. The following example shows you a new self-defined function (this will be discussed in lesson 7) and its use:

```
centre <- function(x, type) {
 switch(type,
        mean = mean(x),
        median = median(x))
}
set.seed(seed = 5) # each time the same pseudo random values
x <- rnorm(10) # draw 10 random values from std. normal distribution
centre(x, "mean") # calculate the mean of x
```

```
#> [1] -0.07885155
```

```
centre(x, "median") # calculate the median of x
```

```
#> [1] -0.37897
```

> **i** Note: Drawing random numbers
>
> When random numbers are drawn with a computer, then these numbers are not complete random for they depend on the initialization (so-called pseudo random numbers).
>
> To ensure that always the same random numbers are drawn, you can fix the initialization with the *set.seed()* function in R.

6.1.3.3 `for` loop

Loops are used in programming to repeat a specific block of code. The `for` loop takes an iterator variable and assigns it successive values from a sequence or vector. `for` loops are most commonly used for iterating over the elements of an object (lists, vectors, *etc.*). Figure 6.3 shows the flowchart of the `for` loop.

A simple example of the use of a `for` loop:

```
for (i in 1:10) {
  print(i)
}
```

This loop takes the ***i*** variable and in each iteration of the loop gives it values 1, 2, 3,..., 10, and then prints the value of ***i*** in the console. After the last value of ***i***, being 10, it exits the loop.

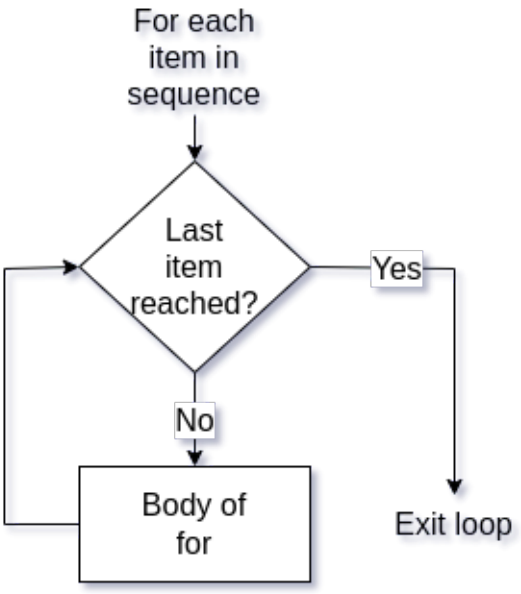


Figure 6.3: `for` loop flowchart

The following three loops have the same behavior (remember the swirl exercise about sequences?).

```
x <- c("a", "b", "c", "d")

# 1)
for (i in 1:4) {
  print(x[i])
}
# or
for (i in 1:4) print(x[i])

# 2)
for (i in seq_along(x)) {
  print(x[i])
}

# 3)
for (letter in x) {
  print(letter)
}
```

It is **perfectly valid** in R to have nested `for` loops.

```
x <- matrix(data = 1:6, nrow = 2, ncol = 3)
# print values in matrix x per row column-wise
for (i in seq_len(nrow(x))) {
  for (j in seq_len(ncol(x))) {
    print(x[i, j])
  }
}
```

Be careful with nesting though. Nesting beyond 2–3 levels is often very difficult to read/understand.

6.1.3.4 `while` loop

The `while` loop begins by testing a condition. If it is true, the loop body is executed. Once the loop body is executed, the condition is tested again, and so forth. Figure 6.4 displays the flowchart for the `while` loop.

For example:

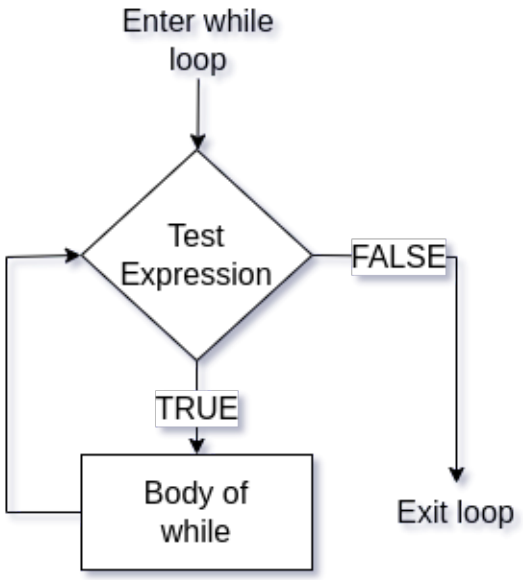


Figure 6.4: `while` loop flowchart

```
counter <- 0
while (counter < 10) {
  print(counter)
  counter <- counter + 1 # raise counter by 1
}
```

> ⚠ Warning: Infinite loops
>
> `while` loops can potentially result in **infinite** loops if not written properly.
>
> **Therefore, use with care!**

6.1.3.5 `break` and `next` statement

In R programming a normal looping sequence can be altered by using the `break` or the `next` statement.

A `break` statement is used inside a loop (`for`, `while`, and `repeat`) to stop the iterations and flow the control outside of the loop. In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated. Figure 6.5 shows the flowchart of the use of the `break` statement.

An example of how to use the `break` statement:

```
x <- 1:5
for (val in x) {
  if (val == 3) {
    break
  }
  print(val)
}
```

In this example, you iterate over the vector object *x*, which has consecutive numbers from 1 to 5. Inside the `for` loop you have used an `if` condition, to break if the current value is equal to 3. The loop terminates, when it encounters the break statement.

A `next` statement is useful, when you want to skip the current iteration of a loop without terminating it. On encountering `next`, the R parser skips further evaluation and starts the next iteration of the loop. Figure 6.6 displays the flowchart for the use of the `next` statement.

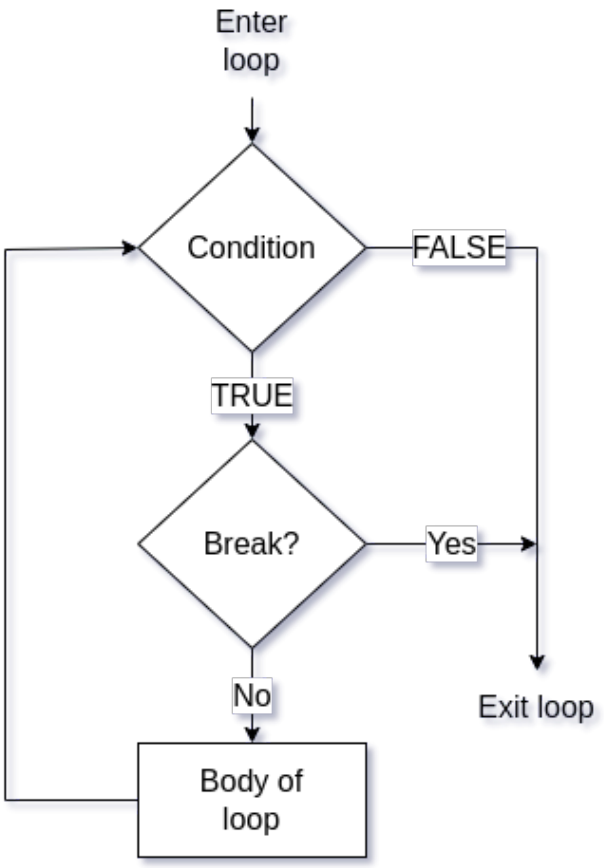


Figure 6.5: flowchart for the use of the `break` statement

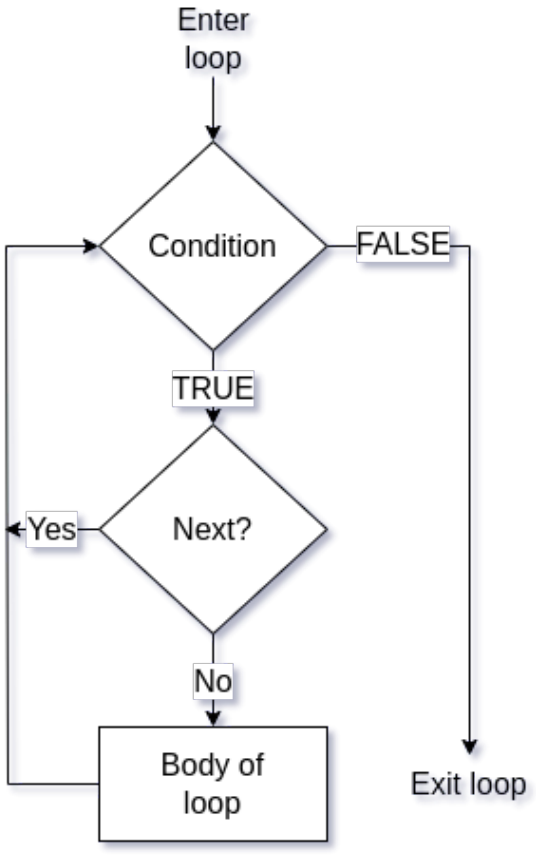


Figure 6.6: flowchart for the use of the `next` statement

An example of using the `next` statement:

```
x <- 1:5
for (val in x) {
  if (val == 3) {
    next
  }
  print(val)
}
```

In the above example the `next` statement is used inside a condition, to check if the value is equal to 3. If the value is equal to 3, the current evaluation stops (value is not printed) but the loop continues with the next iteration.

6.1.3.6 `repeat` loop

The `repeat` statement initiates an **infinite** loop. These are not commonly used in statistical applications, but do have their uses. There is no condition check in a `repeat` loop to exit the loop. The only way to exit a `repeat` loop is to use a `break` statement somewhere to exit the loop. Figure 6.7 shows the flowchart for using a `repeat` loop.

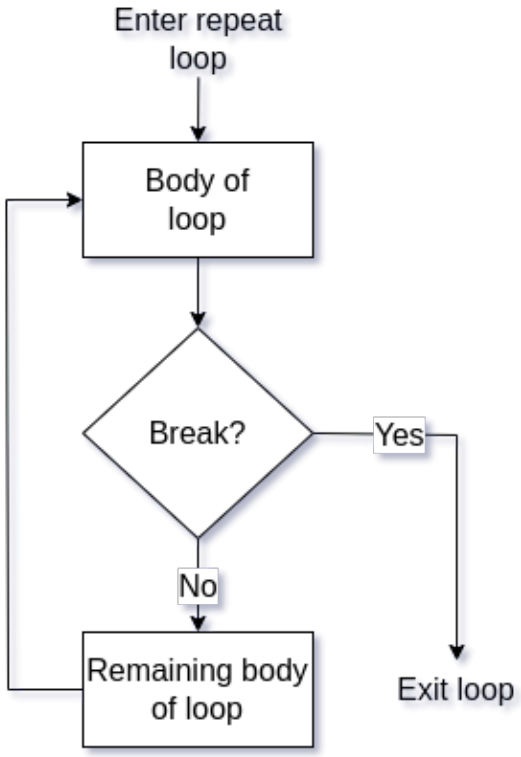


Figure 6.7: `repeat` loop flowchart

A first simple example for a `repeat` loop:

```
counter <- 1 # initialize a counter
repeat {
  print(x)
  counter <- counter + 1 # raise the counter by 1
  if (counter == 6) {
    break
  }
}
```

In the above example a condition to check and exit the loop is used. The loop exits, when the ***counter*** object takes the value of 6.

A more complicated example of using a `repeat` loop, that immediately shows the danger of using this type of looping:

```
x0 <- 1 # initialize a target value
tol <- 1e-8 # set a tolerance for the diffence to the target value
repeat {
  x1 <- computeEstimate()
  if (abs(x1 - x0) < tol) {
    break
  } else {
    x0 <- x1 # update the target value
  }
}
```

Can you see the danger here? Let me explain. The, above provided, `repeat` loop example is a bit dangerous, because there is no guarantee it will stop, i.e. that it will converge. It is better to set a hard limit on the number of iterations (e.g. using a `for` loop) and then report whether convergence was achieved or not.

6.1.4 Exercise: Control Structures

Read the examples given in Section 6.1.3 above about Control Structures. Do you understand what is happening? Next execute these examples, when possible, and verify your output.

6.2 ANOVA and Pairwise Comparison

For theory see O&L Sections 8.1, 8.2, 8.3, 8.4, 9.5, 14.1, 14.2, 14.3, 14.4, 14.5 [5] (6^{th} Edition [4]: additionally Section 9.4 about Fisher's LSD.)

Note: Levene's Test for Homogeneity of variances as an example of One-way ANOVA

Levene's Test

A homogeneity of variances test, that is less dependent on the assumption of normality than most tests, is Levene's Test. For each case, it computes the absolute difference between the value of that case and its cell mean and performs a one-way analysis of variance on those differences.

Assumptions:

1. The samples from the populations under consideration are independent.
2. The populations under consideration are approximately normally distributed.

Checking the assumptions

1. Make sure that the samples were taken independently of one another by assessment of the experimental description.
2. Construct Q-Q plots by treatment to assess normality, and when necessary, when the plots made look like normal distributions, follow up with a test for normality.

Notation (One-way ANOVA)

- $t =$ number of treatments
- $y_{ij} =$ sample observation j for treatment i ($j = 1, 2, \ldots, N_i$ and $i = 1, 2, \ldots, t$)
- $N_i =$ number of observations for treatment i (at least one N_i must be 3 or more)
- $N = N_1 + N_2 + \cdots + N_t =$ total number of observations (overall size of combined samples)
- $\bar{y}_{i\cdot} =$ mean of sample observations from treatment i

- $Z_{ij} = |y_{ij} - \bar{y}_{i.}|$ = absolute deviation of observation j from the treatment i mean
- $Z_{i\cdot} = \frac{1}{N_i}\sum_{j=1}^{N_i} Z_{ij}$ = mean of the Z_{ij} for treatment i
- $Z_{\cdot\cdot} = \frac{1}{N}\sum_{i=1}^{t}\sum_{j=1}^{N_i} Z_{ij}$ = mean of all Z_{ij}

Levene's Testing Procedure

Definition of the parameters: σ_i^2 = population variance for treatment i, where $i = 1, 2, \ldots, t$

Step 1 H_0: $\sigma_1^2 = \sigma_2^2 = \ldots = \sigma_t^2$ vs. H_a: at least one treatment population variance differs from the rest.
Step 2 Test Statistic

$$F = \frac{MST}{MSE} = \frac{\frac{SST}{t-1}}{\frac{SSE}{N-t}} = \frac{\frac{\sum_{i=1}^{t} N_i \times (Z_{i.} - Z_{..})^2}{t-1}}{\frac{\sum_{i=1}^{t}\sum_{j=1}^{N_i}(Z_{ij} - Z_{i.})^2}{N-t}}$$

Step 3 Under H_0 the test statistic $F \sim F-$distribution with $df_1 = t - 1$ and $df_2 = N - t$ degrees of freedom.
Step 4 Under H_a the test statistic F will tend to **higher** values than under the null hypothesis H_0.
Step 5 The test requires the calculation of a right-sided P-value or the determination of a right-sided rejection region to draw the conclusion.
Step 6 Calculate the outcome of the test statistic as defined in **Step 2**.

NOTE: At least one of the treatments must have 3 or more observations or else Levene's test statistic will be undefined (the denominator will equal zero if each treatment has 1 or 2 observations).

Step 7 Calculate the right-sided P-value based on the outcome of the test statistic, or determine the right-sided rejection region based on the chosen level of significance α, both using the F-distribution as defined in **Step 3**.
Step 8 Draw the statistical conclusion in terms of "Reject H_0, and accept H_a." or "Do not reject H_0, and do not accept H_a." based on the calculated P-value in comparison to the chosen level of significance α or based on whether the outcome of the test statistic lies in the rejection region. Also give the conclusion in words starting with "It has been shown (when $\alpha = \ldots$), that <H_a in words>." or "It has not been shown (when $\alpha = \ldots$), that <H_a in words>.".

Example in R code

The example given below clearly shows, that Levene's Test for Homogeneity of variances with two groups (a.k.a. Levene's Test for Equality of variances) gives the same results as doing a one-way ANOVA on transformed observations (absolute deviations of observations from treatment means).

```
sample1 <- c(14, 15, 15, 15, 16, 18, 22, 23, 24, 25, 25)
sample2 <- c(10, 12, 14, 15, 18, 22, 24, 27, 31, 33, 34, 34, 34)

# Levene's test using the car package
dataset <- data.frame(y = c(sample1, sample2),
                      group = as.factor(c(rep.int("sample1",
                                                  times = length(sample1)),
                                          rep.int("sample2",
                                                  times = length(sample2)))),
                      z = c(abs(sample1 - mean(sample1)),
                            abs(sample2 - mean(sample2))))
car::leveneTest(y ~ group, data = dataset, center = mean)
```

```
#> Levene's Test for Homogeneity of Variance (center = mean)
#>       Df F value   Pr(>F)
#> group  1  8.7357 0.007307 **
#>       22
```

```
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
# One-way ANOVA on the absolute differences from the group means
anova(lm(z ~ group, data = dataset))
```

```
#> Analysis of Variance Table
#>
#> Response: z
#>           Df  Sum Sq Mean Sq F value   Pr(>F)
#> group      1  83.973  83.973  8.7357 0.007307 **
#> Residuals 22 211.479   9.613
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Brown-Forsythe Test

Levene's Test for Homogeneity of variances uses: $Z_{ij} = |y_{ij} - \bar{y}_{i\cdot}|$ = absolute deviation of observation j from the treatment i **mean**. This is specified in the *leveneTest()* function with the function argument `center = mean`.

When specifying the function argument as `center = median` in the *leveneTest()* function, you actually perform the Brown-Forsythe Test for Equality of variances [14]. This is a one-way ANOVA on transformed observations, where $Z_{ij} = |y_{ij} - \tilde{y}_{i\cdot}|$ = absolute deviation of observation j from the treatment i **median**.

Although the optimal choice depends on the underlying distribution, the definition based on the median is recommended as the choice that provides good robustness against many types of non-normal data while retaining good statistical power. If one has knowledge of the underlying distribution of the data, this may indicate using one of the other choices. Brown and Forsythe performed Monte Carlo studies that indicated that using the trimmed mean performed best when the underlying data followed a Cauchy distribution (a heavy-tailed distribution) and the median performed best when the underlying data followed a chi-squared distribution with four degrees of freedom (a heavily skewed distribution). Using the mean provided the best power for symmetric, moderate-tailed, distributions.

When the F-test in one-way ANOVA is significant, the conclusion will be that there are differences in mean response between (some of) the treatments. You then need to find out more about these differences. For instance, with three treatments 1, 2 and 3, you want to know whether all three means are different, or 1 and 2 may be the same, but different from 3, or. . .. Therefore, treatments are compared pairwise, i.e. 1 versus 2, 1 versus 3 and 2 versus 3. If for each comparison a t-test is used, each pairwise comparison involves a probability of a Type I Error of size α to wrongly conclude that the two treatments have different mean response. To protect yourself against too many such mistakes, you can use other methods than multiple t-tests. There are a (large) number of such methods for multiple comparisons. Two of them have been discussed in previous statistics courses: Fisher's protected LSD and Tukey's method.

A method that is often used for pairwise comparisons is Fisher's protected LSD method. Oddly enough this method is not discussed in the 7^{th} Edition of O&L [5]. Therefore, it will be presented here.

The method is due to famous statistician Sir Ronald Aylmer Fisher (1890-1962). With Fisher's Protected LSD method, t-tests are carried out for comparing means of each pair of treatments (employing the pooled estimate $s_p^2 = \hat{\sigma}_\epsilon^2 = MSE$ at all times), provided that the overall F-test for treatments is significant. When the F-test is not significant, you simply stop, and no pairwise comparisons are made.

Suppose that all sample sizes for all treatments are the same and equal to n, i.e. each treatment has the same number of repetitions n . The t-test comparing e.g. treatments 1 and 2 is significant, when the absolute value of the test statistic exceeds a critical value, say t_0, from the Student t-distribution:

$$|(\bar{y}_{1\cdot} - \bar{y}_{2\cdot})/\sqrt{\frac{MSE}{n} + \frac{MSE}{n}}| > t_0$$

This is equivalent to $|(\bar{y}_{1.} - \bar{y}_{2.})| > t_0 \times \sqrt{\frac{MSE}{n} + \frac{MSE}{n}}$ or $|(\bar{y}_{1.} - \bar{y}_{2.})| > t_0 \times \sqrt{\frac{2 \times MSE}{n}}$.

Since the expression $t_0 \times \sqrt{\frac{2 \times MSE}{n}}$ of the right operand is the same for any pair of treatments, you can say that any two treatments with an absolute difference between their sample means larger than $t_0 \times \sqrt{\frac{2 \times MSE}{n}}$ differ significantly from each other (according to a t-test against a two-sided alternative hypothesis, employing a pooled variance).

The outcome of $t_0 \times \sqrt{\frac{2 \times MSE}{n}}$ is called the least significant difference or LSD, so:

$$LSD = t_0 \times \sqrt{\frac{2 \times MSE}{n}}$$

When sample sizes are different, e.g. n_1 and n_2 for treatments 1 and 2, the LSD may differ between pairs of treatments, e.g. for treatments 1 and 2:

$$LSD_{12} = t_0 \times \sqrt{\frac{MSE}{n_1} + \frac{MSE}{n_2}} = t_0 \times \sqrt{MSE \times (\frac{1}{n_1} + \frac{1}{n_2})}$$

Note that t_0 only depends upon the number of degrees of freedom corresponding to the mean square error MSE and the probability level (size of the test) α.

6.2.1 Exercise One-way ANOVA: Orange Trees

You will compare the yields (in pounds) of five different varieties (A, B, C, D and E) of 4-year-old orange trees in an orchard. From each variety 7 trees are randomly sampled from this orchard.

Model for the observations:

$$y_{ij} = \mu_i + \varepsilon_{ij}$$

with:
$i = 1,\ldots, 5$; $j = 1,\ldots,7$ and ε_{ij}'s are independent random drawings from N(0, σ).
y_{ij} is the j-th observation on yield for the i-th variety.
μ_i is the expected yield for the i-th variety.

a. Read the data file `orange_tree_yield.csv` into R and assign it the object name ***orange***. Show the content, to get a feeling about what data is contained in the file, you read into R.

b. What are the experimental units in this experiment?

c. Conduct an analysis of variance to the test if the yield for the five varieties could all have the same (population) mean and print the ANOVA output in the console with:

```
# create factor variable of variety
orange$variety <- as.factor(orange$variety)
# build the model
lmOrange <- lm(y ~ variety, data = orange)
# print the output of the ANOVA
anova(object = lmOrange)
```

d. Give the null- and alternative hypothesis (H_0 vs. H_a), the outcome of the test statistic, distribution of the test statistic, p-value and your conclusion in words.

e. Use a `for` loop to calculate the mean of each variety.

f. Make a table with the treatment means, minimum and maximum value for all varieties and the standard deviations (not pooled) using the **doBy** package:

```
library(doBy)
summaryBy(y ~ variety, data = orange, FUN = c(mean, sd, min, max))
```

g. In the next question you will use the **emmeans** package. Assuming that this new package has a function with the same name as defined in the **doBy** package, you used in the previous question. Give the console

command to unload the **doBy** package, to make sure no unexpected behavior will occur when calling that specific function.

h. Make a table with the treatment means for all varieties and the associated standard errors using the **emmeans** package:

```
library(emmeans)
emmeans(lmOrange, ~ variety)
```

i. Give the conclusions for the comparisons of mean yield between pairs of variety, based on pairwise comparisons with t-tests, a.k.a. Fisher's protected LSD , use:

```
# F-test from ANOVA is significant, so continue with pairewise comparisons
pairwise.t.test(orange$y, orange$variety, p.adjust = "none", pool.sd = TRUE)
```

j. Calculate Tukey's W (= difference pronounced significant according to the Tukey method) using $\alpha = 0.05$, and give the results of the pairwise comparison procedure using this W.

The **agricolae** package contains a number of functions for multiple comparisons:

```
library(agricolae)
# ordinary LSD (a.k.a. Fisher's protected LSD)
LSD.test(lmOrange, "variety", console = TRUE)
# Bonferroni
LSD.test(lmOrange, "variety", p.adj = "bonferroni", console = TRUE)
# Tukey
HSD.test(lmOrange, "variety", group = TRUE, console = TRUE)
```

k. Check the model assumptions for ANOVA by making a Q-Q plot of the residuals and a residual vs. predicted values plot. Hint: See Section 4.2.3.

6.2.2 Exercise Two-way ANOVA: City Health

In a few cities in the Netherlands a sample is taken from healthy and not-so-healthy elderly people. After some investigations (interview), a team of psychologists determines a score for the extent to which they suffer from a depression. Variable city is a factor with 4 levels: 1 = Amsterdam, 2 = Maastricht, 3 = Ede, 4 = Groningen. Variable health is a factor with 2 levels: 1 = good, 2 = poor.

a. Read the data file `city_health_score.csv` into R and assign it to an object name ***depression***.

b. Make sure that city and health are factors and assign the appropriate level names to these factors.

c. Make a meaningful graph of the data; use the *plot()* function and also try the function *interaction.plot()*.

d. Perform an analysis of variance to test for effect of city, health and the interaction. What is/are your conclusion(s)?

e. Give descriptive statistics (group means, for factor levels and factor level combinations); try to use the *summaryBy()* function from the **doBy** package.

f. Give parameter estimates, using the function *coef(**lmObject**)* (for ***lmObject*** fill the object name you have assigned). Use them to estimate the mean score for "Ede, good health".

g. Use the *emmeans()* function from the **emmeans** package to find the mean score for "Ede, good health" with its associated standard error.

h. Check your model assumptions. [Hint: Remember the function *plot(**lmObject**)*, filling your own object name for ***lmObject***.]

6.3 Command Reference

Table 6.1: Lesson 6 commands

Command	Description
%%	modulo operator, remainder of x/y
anova()	compute ANOVA table for fitted model objects
`break`	control within a loop: exit the loop
`for`	for loop statement initiator
HSD.test()	make multiple comparisons Tukey (**agricolae** package)
`if...else`	`if...else` statement initiator
ifelse()	conditional element selection
interaction.plot()	two-way interaction plot
LSD.test()	multiple comparisons (**agricolae** package)
emmeans()	least-squares (or estimated marginal) means (**emmeans** package)
`next`	control within a loop: skip to next loop
old.packages()	check for available package updates
pairwise.t.test()	calculate pairwise comparisons for multiple testing
`repeat`	repeat loop statement initiator
require()	load and attach add-on packages
set.seed()	initiate a starting point for pseudo random number generation
summaryBy()	calculate group wise summary statistics (**doBy** package)
switch()	select one of a list of alternatives
update.packages()	download and install available package updates
`while`	while loop statement initiator

Lesson 7

Functions and Multiple Regression

7.1 Functions in R

As was explained earlier, functions are the basic building blocks in R to get things done. John M. Chambers, the creator of the S programming language [15], and core member of the R programming language, once said:

- "Everything that exists, is an object." and,
- "Everything that happens, is a function call.".

R has many, many built-in functions. In this lesson you are going to look at R functions more carefully. Not only will you see how existing functions can be called, but also how easy it is to make your own! Actually, this is what people do if they make new packages: they program a new set of functions and spread them worldwide in an R package.

7.1.1 Exercise SWIRL: Functions

First you are going to use the **swirl** package to study functions.

Start up SWIRL by loading the package and starting it:

```
library(swirl)
swirl()
```

Choose the **R Programming** course, followed by Lesson 9: Functions.

> ⚠ Warning: Answers in SWIRL
>
> Keep in mind that answers in SWIRL Lessons are sensitive to the smallest of errors, so follow the instructions given in a lesson carefully!

In this lesson you will see:

- structure of a function: name, arguments between round brackets, body between curly brackets
- typing the name of function without brackets shows the source code
- last expression evaluated is returned by function
- default arguments for functions so that some arguments need not be specified in call
- function calls with explicitly names arguments, so that order is unimportant
- abbreviating names of arguments
- function *args()* to see arguments of a function
- functions as arguments to other functions
- *anonymous* functions
- ellipses . . . argument for an indefinite number of arguments
- defining binary operators like `+`, `-`, `*`, `/`

7.1.2 Function Control Structure: The *return()* Function

In SWIRL Lesson 9 about Functions, you have just worked through, you have learned that the last expression evaluated in a function gets returned by the function. There is, however, also a control structure available, that will give you more control on what, and where objects get returned inside a function.

The *return()* function is used inside a new function definition and signals that a function should exit and return a given value.

An example of the use of *return()* inside a function definition and its use:

```
check <- function(x) {
  if (x > 0) {
    result <- "Positive"
  } else if (x < 0) {
    result <- "Negative"
  } else {
    result <- "Zero"
  }
  return(result)
}
# some sample runs.
check(1)
```

```
#> [1] "Positive"
```

```
check(-10)
```

```
#> [1] "Negative"
```

```
check(0)
```

```
#> [1] "Zero"
```

If there are no explicit returns from a function, as you have seen in the SWIRL Lesson 9, the value of the last evaluated expression is returned automatically in R.

For example, the following is equivalent to the above function:

```
check <- function(x) {
  if (x > 0) {
    result <- "Positive"
  } else if (x < 0) {
    result <- "Negative"
  } else {
    result <- "Zero"
  }
  result
}
```

Generally explicit use of the *return()* function to return a value immediately within a new function definition is advised. Mainly for readability of your code.

If *return()* is not the last statement of the new function definition, it will prematurely end the function bringing the control to the place from which it was called. For example:

```
check <- function(x) {
  if (x > 0) {
    return("Positive")
  } else if (x < 0) {
    return("Negative")
  } else {
    return("Zero")
  }
}
```

In the above example, `if x > 0`, the function immediately returns "Positive" without evaluating rest of the body.

The *return()* function can return only a single object. If you want to return multiple values in R, you need to use a list (or other objects) and return it.

Following is an example:

```
multi_return <- function() {
  my_list <- list("color" = "red",
                  "size" = 20,
                  "shape" = "round")
  return(my_list)
}
a <- multi_return()
a$color
```

```
#> [1] "red"
```

```
a$size
```

```
#> [1] 20
```

```
a$shape
```

```
#> [1] "round"
```

Here a list is created, named ***my_list*** inside the function *multi_return()*, with multiple elements and this single list is returned by the function named *multi_return()* and assigned to an object named ***a***.

7.1.3 Example function: Standard Error of the Mean (s.e.m.)

As an example we make a function to calculate the standard error of the mean. This function is not available within basic R.

Suppose we have a random sample $y_1, \ldots, y_n$ from some population with population mean μ and standard deviation σ. As estimator of μ we use the sample mean $\bar{y}$. As estimator of σ we take the sample standard deviation s, available in R with function *sd()*.

For hypothesis testing or confidence intervals for μ we need (an estimator of) the standard error of the mean: $sem = s/\sqrt{n}$. Below a function *sem()* is defined. It takes as argument a numerical vector y and returns the $sem(y)$. Try the code yourself in R.

```
sem <- function(y) {
  n <- length(y)
  result <- sd(y) / sqrt(n)
  return(result)
}
```

Apply this function to some data. We sample 25 values from a standard normal distribution ($\mu = 0$, $\sigma = 1$). The standard error of the mean should approximately be around $1/\sqrt{25} = 0.2$. Is this what you see below?

```
set.seed(seed = 1492) # ensure same pseudo random numbers for everyone
z <- rnorm(n = 25) # draw 25 numbers from N(0, 1)
mean(z) # calculate the mean
```

```
#> [1] -0.2579794
```

```
sd(z) # calculate the standard deviation
```

```
#> [1] 0.9903985
```

```
length(z) # number of observations in vector z
```

```
#> [1] 25
```

```
sem(z) # calculate the standard error of the mean
```

```
#> [1] 0.1980797
```

When random numbers are drawn with a computer, then these numbers are not complete random for they depend on the initialization (so-called pseudo random numbers). To ensure that always the same random numbers are drawn you can fix the initialization with the *set.seed()* function in R.

7.1.4 Exercise: Functions, Doing It Yourself (DIY)

Make an R function that takes three arguments:

1. a vector with numeric data to be summarized (possibly containing missing values)
2. a number of decimals to be printed; the default number should be 2
3. a logical telling whether missing values should be removed prior to calculating means and standard deviations; the default value should be `TRUE`.

The function should return a string, containing the mean and standard deviation in the form `mean (standard deviation)`, like 10.21 (2.13). [Hint: the function *round()* can be used to get the desired number of decimals; the function *paste()* can be used to create the desired string.]

7.2 Multiple regression

For theory see O&L Sections 12.1, 12.2, 12.3, 12.4, 12.5, 12.6, 7th Edition [5] Sections 13.2 pp.718-719, 13.4 p.745 and p.753 (6th Edition [4]: Sections 13.2 pp.770-771, 13.4 p.797 and p.805).

The multiple linear regression model describes a linear relationship between dependent variable y and *multiple* regressors (explanatory variables) $x_1, \ldots, x_k$:

$$y_i = \beta_0 + \beta_1 x_{1i} + \ldots + \beta_k x_{ki} + \varepsilon_i \tag{7.1}$$

with $i = 1, \ldots, n$ as an index for the observation number. As in simple linear regression, we assume that the errors ε_i are independent, have constant variance and are normally distributed with expected value 0.

Least-squares estimates of the parameters β_0, β_1, ..., β_k are obtained using the R function *lm()*, like we did in simple linear regression. This function wants a model formula `y ~ x1 + ... + xk` as input. The results from the linear regression are saved into an R object of class lm. All functions shown in Lesson 4 on Simple Linear Regression can be used for multiple regression again.

Below we look at an example about yield of an orchard (y) and its possible dependency upon the average January (x_1) and May (x_2) temperatures. Observations have been collected during six consecutive years. The following multiple linear regression model is assumed: $y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \epsilon_i$ with $i = 1 \ldots 6$.

Please type the commands shown below into an R script, or, even better, into an Rmd file.

```
y  <- c(31.4, 32.0, 32.3, 33.2, 32.7, 33.4)
x1 <- c( 0.5,  1.0,  2.0,  2.5,  3.0,  3.0)
x2 <- c(12.0, 12.0, 13.0, 14.0, 13.0, 14.0)

lmo <- lm(y ~ x1 + x2) # fit the multiple regression model
summary(object = lmo)  # summary of results
```

```
#>
#> Call:
#> lm(formula = y ~ x1 + x2)
#>
#> Residuals:
#>       1       2       3       4       5       6
#> -0.16667  0.30000 -0.20000  0.03333 -0.06667  0.10000
#>
#> Coefficients:
```

```
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  25.0333     2.6667   9.387  0.00256
#> x1            0.2667     0.1963   1.359  0.26737
#> x2            0.5333     0.2301   2.317  0.10332
#>
#> Residual standard error: 0.2404 on 3 degrees of freedom
#> Multiple R-squared:  0.939,  Adjusted R-squared:  0.8983
#> F-statistic: 23.08 on 2 and 3 DF,  p-value: 0.01508
```

```
confint(object = lmo)  # confidence intervals for parameters
```

```
#>                  2.5 %     97.5 %
#> (Intercept) 16.5465889 33.5200778
#> x1          -0.3579245  0.8912579
#> x2          -0.1990648  1.2657315
```

The F-test shown at the bottom of the output from *summary()* is the, so-called, "omnibus" F-test. It tests $H_0: \beta_1 = \beta_2 = 0$. In words, it tests whether there is any explanatory power at all for this regression model.

We continue with some assumption checking, as shown in Figure 7.1. Notice the function *rstandard()* to obtain standardized residuals.

```
r <- residuals(object = lmo) # ordinary residuals
sr <- rstandard(model = lmo) # standardized residuals
f <- fitted(object = lmo)    # fitted values

# normal Q-Q plot using the standardized residuals
qqnorm(sr); qqline(sr, col = "red")

# residual plot; too few points to say anything about model assumptions:
plot(r ~ f,
     main = "Check constant variance",
     ylab = "residuals",
     xlab = "fitted values")
# add horizontal line residuals = 0 in residual plot
abline(h = 0)
```

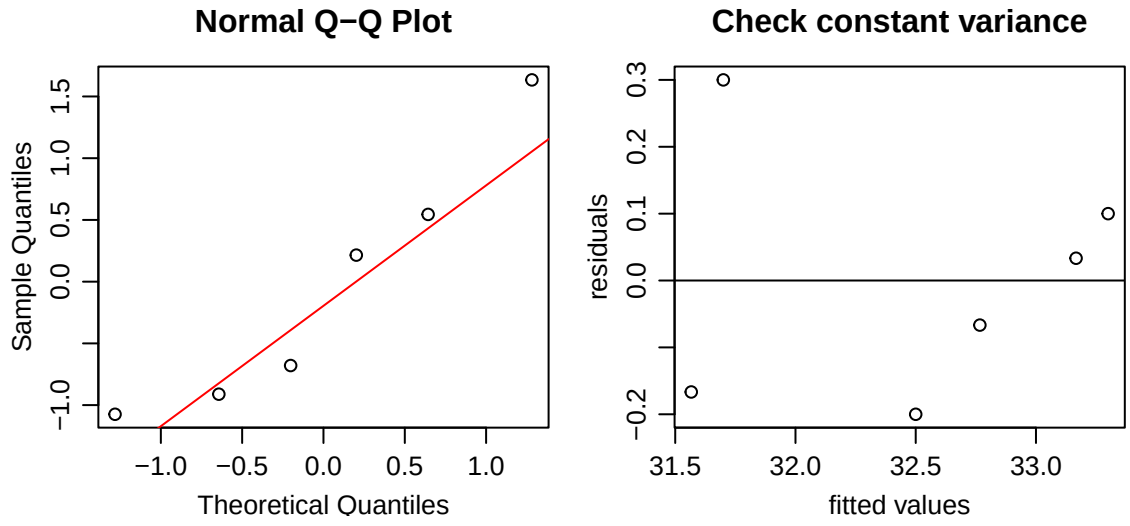


Figure 7.1: Plots to check the assumptions for multiple linear regression

We conclude with estimation of the mean response at new values of the regressors, including a confidence interval. Here we want to estimate the mean yield if the January temperature is 1°C and May temperature is 14°C.

```
newdata <- data.frame(x1 = 1, x2 = 14)
predict(object = lmo,
```

```
        newdata = newdata,
        se.fit = TRUE,
        interval = "confidence")
```

```
#> $fit
#>        fit      lwr     upr
#> 1 32.76667 31.42343 34.1099
#>
#> $se.fit
#> [1] 0.422076
#>
#> $df
#> [1] 3
#>
#> $residual.scale
#> [1] 0.2403701
```

In simple linear regression we did not discuss the ANOVA table. The ANOVA table is a table, that tells how well the regression model is fitting, by comparing the multiple linear regression model with the simplest model possible, i.e. the model with only a constant (intercept). The residual sums of squares of these two models (which are the products of the least-squares procedure) are put in a table, together with the difference between the two. This difference (the model sum of squares) tells how much of the variation is explained by the regressors.

In R a "sort of" ANOVA table is obtained using the function *anova()*. First we are going to look at this function. Later we are going to modify it by creating a new function *newAnova()*, using the knowledge you gained in the previous Section 7.1.

```
anova(object = lmo)
```

```
#> Analysis of Variance Table
#>
#> Response: y
#>           Df  Sum Sq Mean Sq F value   Pr(>F)
#> x1         1 2.35636 2.35636 40.7832 0.007775
#> x2         1 0.31030 0.31030  5.3706 0.103324
#> Residuals  3 0.17333 0.05778
```

The call `anova(object = lmo)` produces an "Analysis of Variance Table", a.k.a ANOVA table. In the column `Sum Sq` sums of squares are shown.

At the bottom of the ANOVA table the residual sum of squares is presented, which is the result from the least-squares procedure. This is the value that can be obtained with the *deviance()* function.

```
deviance(object = lmo)
```

```
#> [1] 0.1733333
```

Further up in the table you see the contribution of each regressor to the model sum of squares. These are so-called sequential sums of squares, obtained by fitting a sequence of models. Let's try to reproduce them, so that you understand how they are constructed.

Start with a model with only an intercept, and store the residual sum of squares of this model. Notice that we have to supply `1` as the right-hand-side of the model formula, to tell to R that we want a model with intercept only.

```
M0 <- lm(y ~ 1)
(SSE0 <- deviance(object = M0))
```

```
#> [1] 2.84
```

Next fit a model with intercept and x_1, and store the residual sum of squares of this second model. Notice that specification of the explicit intercept (`1`) is not needed anymore (although it doesn't harm if you would add it):

```
M1 <- lm(y ~ x1)
(SSE1 <- deviance(object = M1))
```

```
#> [1] 0.4836364
```

Calculate the difference in SSE0 and SSE1. Notice that this is exactly the sum of squares for x_1 shown in the ANOVA table earlier.

```
(SS1x1 <- SSE0 - SSE1)
```

```
#> [1] 2.356364
```

Next add x_2 to the model that already contains intercept and x_1, and calculate again the difference in residual sums of squares. This is the sum of squares for x_2 after x_1, as shown in the earlier ANOVA table.

```
M2 <- lm(y ~ x1 + x2)
(SSE2 <- deviance(object = M2))
```

```
#> [1] 0.1733333
```

```
(SS1x2x1 <- SSE1 - SSE2)
```

```
#> [1] 0.310303
```

If we would change the order of terms, we will get a different ANOVA table (!):

```
anova(object = lm(y ~ x2 + x1))
```

```
#> Analysis of Variance Table
#>
#> Response: y
#>           Df  Sum Sq Mean Sq F value   Pr(>F)
#> x2         1 2.56000 2.56000 44.3077 0.006911
#> x1         1 0.10667 0.10667  1.8462 0.267367
#> Residuals  3 0.17333 0.05778
```

Hence, it is **NOT** possible to say which effect is caused by x_1 and which effect is caused by x_2. It is only possible to say what the effect of e.g. x_1 is if we also specify which model it is part of!

The order-dependence of sums of squares is caused by correlation among the regressors. This is called multicollinearity. The collinearity of a regressor with all the other regressors can be measured with the Variance Inflation Factor (VIF). A value for VIF equal to 1 tells that there is no collinearity of the variable with all others. The function *vif()* can be found in the R package **car**.

```
suppressWarnings(library(car)) # suppresses a warning message being printed
vif(mod = M2)
```

```
#>       x1       x2
#> 3.666667 3.666667
```

The *anova()* function can be used to compare two nested models, a Full Model FM and a (smaller) Reduced Model RM as described in O&L Section 12.5 of both the 6th Edition [4] (p. 691) and the 7th Edition [5] (p.652). It produces an F-test (for a subset of regression coefficients). Below we compare the Full Model ***M2*** (FM), containing both x_1 and x_2 with the smaller Reduced Model ***M1*** (RM), containing x_1 alone. In other words, we are testing $H_0: \beta_2 = 0$ versus $H_a: \beta_2 \neq 0$. The test statistic F can be calculated with:

$$F = \frac{(SSE_{RM} - SSE_{FM})/(df_{RM} - df_{FM})}{SSE_{FM}/df_{FM}} \tag{7.2}$$

where SSE represents the sum of squares residuals and df the degrees of freedom of, respectively, the full model (FM) and the reduced model (RM).

```
anova(M1, M2)
```

```
#> Analysis of Variance Table
#>
```

```
#> Model 1: y ~ x1
#> Model 2: y ~ x1 + x2
#>   Res.Df     RSS Df Sum of Sq      F Pr(>F)
#> 1      4 0.48364
#> 2      3 0.17333  1    0.3103 5.3706 0.1033
```

7.2.1 Exercise Multiple Regression: Electricity and Gas Prices in USA

We look at a data set about yearly electricity use per household in each of the 50 states in the USA (around 1990). The aim was to find out how electricity use y is related to the price of electricity (x_1), per capita income (x_2) and gas price (x_3). We assume a multiple linear regression model: $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \varepsilon$.

a. Read the data file electricity_use_USA.csv into R.

b. Make a matrix scatterplot of the 4 variables (use *pairs()*, check `?pairs` on how to use this function).

c. Fit a multiple linear regression model, explaining electricity use y from electricity price (x_1), per capita income (x_2) and gas price (x_3).

d. Test if the model has any explanatory value.

e. Give the estimated regression equation.

f. Give the estimate and standard error of β_3 (the slope for price of gas).

g. Fit a simple linear regression model with x_3 alone, and give the estimate and standard error of the slope of x_3 now.

h. Compare the two values for slopes (from the multiple and simple regression models). What may be the reason for the difference?

i. Calculate the variance inflation factors for the three regressors.

7.2.2 Exercise Functions: Redefining the Regression ANOVA table

In this final, challenging, exercise for Lesson 7 you are going to program a new ANOVA function. Let's call it *newAnova()*. It should modify the ANOVA table produced by the *anova()* function, in such a way that the new table has only three rows: one row for Model, one row for Residual, and one row for Corrected Total.

The columns of the new ANOVA table remain as they are.

The argument for the new function is an object of class lm, as produced by the *lm()* function.

The result of the new function is a numerical matrix with 3 rows and 5 columns. The column headings of the ANOVA table should be: `Df`, `Sum Sq`, `Mean Sq`, `F value`, `Pr(>F)`. These are simply the column names, that can be assigned to the matrix with function *colnames()*. The row names of the ANOVA table should be: `Model`, `Residuals`, and `Corr. Total`. These are simply row names, that can be assigned to the matrix with function *rownames()*.

Now program the new anova function named *newAnova()*. Within the body of the function do the following:

- Run the old *anova()* on the linear model object, and store the ANOVA table e.g. under object name ***oldm***; force this result to become a matrix, using `oldm <- as.matrix(oldm)`, so that the result is just a numerical matrix.
- Determine the number of rows *nr* of this matrix; this will be the number of regressors + 1.
- Create a new matrix with 3 rows and 5 columns.
- Assign the last row of this old matrix to the second row in the new matrix.
- In the old matrix, add degrees of freedom (column 1) and sums of squares (column 2) of row 1 up to (and including) row $nr - 1$; you may use the function *colSums()* for this.
- These accumulated values are the model degrees of freedom and model sum of squares of the new matrix, so put them in the first row (columns 1 and 2) of the new matrix.
- Based upon the degrees of freedom and sum of squares, calculate the mean square, the F-statistic, and the P-value (using the *pf()* function), and enter these numbers in columns 3, 4, and 5 of the first row of the new matrix.

- Fit the intercept-only model, so model formula `y ~ 1` ; for this we need the dependent variable y itself; you can get it out of the linear model object with the extractor function *model.frame()* (assuming that the linear model object has name ***lmo***): `y <- model.frame(lmo)[, 1]`.
- Run the old *anova()* on the intercept-only result and force the result to be a matrix (which has only one row).
- Copy this row into row 3 of the new matrix.
- Add proper column names and row names (see above).
- Print the resulting new matrix

The result for the old and new anova functions, applied to the regression for the orchard data, should look like this:

```
anova(lmo)
```

```
#> Analysis of Variance Table
#>
#> Response: y
#>           Df  Sum Sq Mean Sq F value   Pr(>F)
#> x1         1 2.35636 2.35636 40.7832 0.007775
#> x2         1 0.31030 0.31030  5.3706 0.103324
#> Residuals  3 0.17333 0.05778
```

```
newAnova(lmo)
```

```
#>             Df    Sum Sq    Mean Sq  F value     Pr(>F)
#> Model        2 2.6666667 1.33333333 23.07692 0.01507807
#> Residuals    3 0.1733333 0.05777778       NA         NA
#> Corr. Total  5 2.8400000 0.56800000       NA         NA
```

7.3 Command Reference

Table 7.1: Lesson 7 commands

Command	Description
...	ellipses argument in function definition, see `?dots` for help
function()	define new functions in the R language
model.frame()	extract the model frame from a formula or fit
pairs()	produce a matrix of scatterplots
pf()	distribution function for the $F(df_{num}, df_{denom})$ distribution
return()	explicitly output the object, given as argument, in a function
rstandard()	provide standardized residuals of a fitted model object \%
set.seed()	initialize the start for random number generation
suppressWarnings()	evaluate ignoring all warnings
vif()	calculate variance inflation factors for (generalized) linear models

Lesson 8

Loop Functions and Repeated Measurement Data

8.1 Looping on the command line in the console

Writing `for`, `while`, and, although less common, `repeat` loops is useful, when writing scripts or programming functions, but not particularly easy when working interactively on the command line in the console. There are some functions, which implement looping to make life easier. In this lesson you learn about some of these functions, which all fit into the framework based on the principle [16]:

- Split up some data into smaller pieces
- Apply a function to each piece
- Combine the result

Figure 8.1 nicely visualizes the Split-Apply-Combine principle.

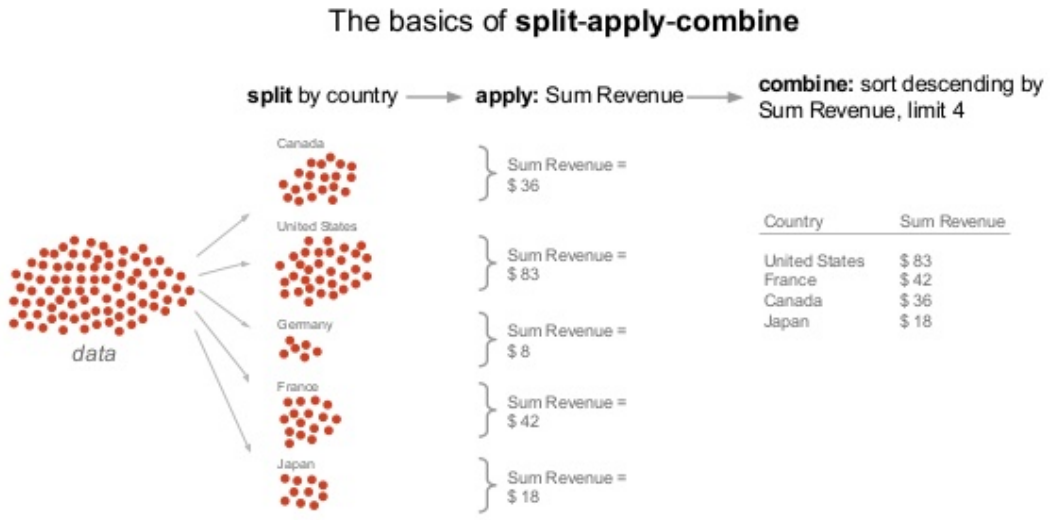


Figure 8.1: Basics of the Split-Apply-Combine principle visualized.

The group of functions, which fit in the above Split-Apply-Combine framework, are sometimes referred to as the **apply()* family of functions, which contains, a.o.:

- *apply()*: Apply a function over the margins of an array.
- *lapply()*: Loop over a list and evaluate a function on each element.

- *sapply()*: Same as *lapply()* but tries to simplify the result. You could read the function name as a s(implify) apply function.
- *tapply()*: Apply a function over subsets of a vector.
- *mapply()*: Multivariate version of lapply.

An auxiliary function *split()* is also useful, particularly in conjunction with *lapply()*, and fits perfectly in the first step of the mentioned framework.

In the following subsections the functions, mentioned above, will be briefly discussed.

8.1.1 The *apply()* function

Let's start of with the godfather of the **apply()* family, which is *apply()* itself. This function takes as input an array (a multi-way array or a data frame), and evaluates a function over the margins of the array (margin = row or column). When the *apply()* function is used, it applies a specific, or self-defined function to the rows, when margin equals 1 (rows), or to the columns, when margin equals 2, or both (rows and columns). When applying a function to either margin 1 (the rows of the array) or margin 2 (the columns of the array), *apply()* returns a vector.

It is most often used to apply a function to the rows or columns of a matrix, but can be used with a general array, e.g. taking the mean of an array of matrices. Using the *apply()* function is not really faster than writing a loop, but it works in one line and takes less space in your code. Usually it takes a little while to get accustomed to writing and understanding the **apply()* family of functions.

The arguments of the *apply()* function are (check the help page for what they mean):

```
str(apply)
```

```
#> function (X, MARGIN, FUN, ..., simplify = TRUE)
```

Examples of using *apply()*:

```
# ensure same pseudo random numbers, when copied
set.seed(seed = 1492)
# create a 3 by 5 matrix with drawings from N(100, 30)
(dat <- matrix(data = rnorm(n = 15,
                            mean = 100,
                            sd = 30),
               nrow = 3,
               ncol = 5))
```

```
#>          [,1]      [,2]      [,3]      [,4]      [,5]
#> [1,] 82.30457 131.43765  74.20412 112.44253  59.84284
#> [2,] 73.04028  91.65134 101.62645 110.49652 106.38743
#> [3,] 51.27667 110.87479  65.25379  66.57651  88.11110
```

```
# find mean of each row in object dat
apply(X = dat, MARGIN = 1, FUN = mean)
```

```
#> [1] 92.04634 96.64040 76.41857
```

```
# or equivalently
rowMeans(x = dat)
```

```
#> [1] 92.04634 96.64040 76.41857
```

```
# find sum of each column in dat
apply(X = dat, MARGIN = 2, FUN = sum)
```

```
#> [1] 206.6215 333.9638 241.0844 289.5156 254.3414
```

```
# or equivalently use
colSums(x = dat)
```

```
#> [1] 206.6215 333.9638 241.0844 289.5156 254.3414
```

The equivalent column function to calculate means, is called *colMeans()*, and the equivalent row function for sums, is called *rowSums()*. To summarize for a matrix object ***x***:

- *rowSums()* → `apply(X = x, MARGIN = 1, FUN = sum)`
- *rowMeans()* → `apply(X = x, MARGIN = 1, FUN = mean)`
- *colSums()* → `apply(X = x, MARGIN = 2, FUN = sum)`
- *colMeans()* → `apply(X = x, MARGIN = 2, FUN = mean)`

> **i** Note: Speed of Shortcut functions.
>
> The shortcut functions, provided above, are much **faster**, but you will not notice unless you are using a large matrix. In general dedicated functions in R are faster than using **apply()* functions or programming loops yourself, because most of the time dedicated functions have be optimized for the job they need to do.

Other examples of using *apply()*:

```
# ensure same pseudo random numbers, when copied
set.seed(seed = 1492)
# matrix 5 by 10 with values from N(0, 1)
x <- matrix(data = rnorm(n = 50), nrow = 5, ncol = 10)
# quantiles of the rows of matrix x
# transposed (t() function) view
t(apply(X = x, MARGIN = 1, FUN = quantile, probs = c(0.25, 0.50, 0.75)))
```

```
#>             25%         50%       75%
#> [1,] -0.4535091  0.32827336 0.3699455
#> [2,] -0.8889588 -0.40106623 0.5785320
#> [3,] -1.2108242 -0.04586507 0.2816595
#> [4,] -0.0960555  0.48266006 1.0377953
#> [5,] -0.3667948 -0.01550604 0.3925751
```

Try `apply(X = x, MARGIN = 2, FUN = quantile, probs = c(0.25, 0.50, 0.75))` yourself to see the quantiles of the columns of matrix **x**.

```
# ensure same pseudo random numbers, when copied
set.seed(seed = 1492)
# create a 2 by 2 by 10 array (dimensions: i, j, k) with values from N(0, 1)
a <- array(data = rnorm(2 * 2 * 10), dim = c(2, 2, 10))
# average matrix in an array (mean over dimension k)
apply(X = a, MARGIN = c(1, 2), FUN = mean)
```

```
#>            [,1]       [,2]
#> [1,] -0.2431086 -0.5245023
#> [2,]  0.3967627 -0.1791942
```

```
# or equivalently
rowMeans(x = a, dims = 2)
```

```
#>            [,1]       [,2]
#> [1,] -0.2431086 -0.5245023
#> [2,]  0.3967627 -0.1791942
```

8.1.2 Exercise: Application of *apply()* on the *iris* data set

The famous (Fisher's or Anderson's) ***iris*** data set, from the base **datasets** package, gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively for 50 flowers from each 3 species of iris. The species are *Iris setosa*, *versicolor*, and *virginica*. For more info check out the help page with `?iris`.

A good data analyst would first "eyeball" the data. Meaning that you would first check the dimensions, whether there are missing values, what the column names are. In general, you would want to know what data

you are dealing with. This should also be your first step.

Once you get a feeling about the data write an expression, using the *apply()* function, that returns a vector of the means of the variables sepal length and width and petal length and width over all species.

8.1.3 The *lapply()* and *sapply()* function

The *lapply()* function takes three arguments:

1. a list object ***X***,
2. a function (or the name of a function) `FUN`,
3. other arguments via its ellipsis (...) argument.

If ***X*** is not a list, it will be coerced to a list using the *as.list()* function. The *lapply()* function applies the specified function to each element of the list and returns a list object. The "*l*" in *lapply()* can be understood, since the function always returns a list object.

```
str(lapply)
```

```
#> function (X, FUN, ...)
```

Examples of the use of *lapply()*:

```
# 1)
# create character vector with city names
cities <- c("New York", "London", "Cape Town")
# for each element in cities, apply function nchar() to count number of characters
lapply(X = cities, FUN = nchar)
```

```
#> [[1]]
#> [1] 8
#>
#> [[2]]
#> [1] 6
#>
#> [[3]]
#> [1] 9
```

```
# to return values as vector, use unlist:
unlist(lapply(X = cities, FUN = nchar))
```

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

```
# 2)
x <- 3:4
# draw 3 and 4 random values from unif(0, 10) (continuous uniform distribution),
# using the ellipsis argument for specifying the function runif() arguments!
lapply(X = x, FUN = runif, min = 0, max = 10)
```

```
#> [[1]]
#> [1] 6.894328 2.915151 9.019213
#>
#> [[2]]
#> [1] 6.048277 5.800525 9.431590 8.431327
```

lapply() and friends make heavy use of *anonymous* functions (see previous lesson), meaning functions defined without a specific name to use in the function call.

```
(x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2)))
```

```
#> $a
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
#>
```

```
#> $b
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
```

```
# extract second row from each matrix in list x
lapply(X = x, FUN = function(elt) elt[2, ])
```

```
#> $a
#> [1] 2 4
#>
#> $b
#> [1] 2 5
```

The *sapply()* function works similarly as the *lapply()* function. The only difference is that *sapply()* will try to simplify, hence the "s" in *sapply()*, the result of the *lapply()* function if possible:

- If the result is a list where every element is of length 1, then a vector is returned.
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can not figure things out, a list is returned.

```
# ensure same pseudo random numbers, when copied
set.seed(seed = 1492)
# list with 1:4, 10 values from N(0, 1), 20 from N(1, 1) and 100 from N(5, 1)
x <- list(a = 1:4,
          b = rnorm(n = 10),
          c = rnorm(n = 20, mean = 1),
          d = rnorm(n = 100, mean = 5))
# mean over each list element
lapply(X = x, FUN = mean)
```

```
#> $a
#> [1] 2.5
#>
#> $b
#> [1] -0.3529594
#>
#> $c
#> [1] 0.979685
#>
#> $d
#> [1] 5.034091
```

```
# try to simplify the result
sapply(X = x, FUN = mean)
```

```
#>          a          b          c          d
#>  2.5000000 -0.3529594  0.9796850  5.0340907
```

```
# show the class of the result
class(sapply(X = x, FUN = mean))
```

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

8.1.4 Exercise SWIRL: *lapply()* and *sapply()*

Start *swirl()* from the package **swirl**, as you have done before, and work through lesson 10. In this lesson, you'll learn how to use *lapply()* and *sapply()*, the two most important members of R's **apply()* family of functions, also known as loop functions.

8.1.5 The *tapply()* function

The *tapply()* function is used to apply a function over subsets of a vector. It is not clear why the function is called *tapply()*, i.e. no obvious reason to start with the letter "t".

```
str(tapply)
```

```
#> function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

where:

- `X` is a vector
- `INDEX` is a factor or a list of factors (or else they will be coerced into factors). It tells R how to group the elements in the vector
- `FUN` is a function to be applied to each group in the vector
- ... (ellipsis argument) contains other arguments to be passed into `FUN`
- `simplify`, logical indicating whether the result should be simplified

An example of taking group means in a vector:

```
# ensure same pseudo random numbers, when copied
set.seed(seed = 1492)
# create vector x with random values, 10 from N(0, 1), 10 from unif(0, 1),
# and 10 from N(1, 1)
x <- c(rnorm(n = 10), runif(n = 10), rnorm(n = 10, mean = 1))
# create a factor variable f with n = 3 levels, with each levels k = 10 replications
(f <- gl(n = 3, k = 10))
```

```
#>  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
#> Levels: 1 2 3
```

```
# take group means over each level
tapply(X = x, INDEX = f, FUN = mean)
```

```
#>          1          2          3
#> -0.3529594  0.5235485  0.9366296
```

In essence, *tapply()* has taken the mean of all the values in each level (indicated with 1, 2, and 3). The `simplify` argument is by default `TRUE`, but of course it can also be specified differently:

```
tapply(X = x, INDEX = f, FUN = mean, simplify = FALSE)
```

```
#> $`1`
#> [1] -0.3529594
#>
#> $`2`
#> [1] 0.5235485
#>
#> $`3`
#> [1] 0.9366296
```

To find for example group ranges, the `FUN` argument should be given the *range()* function (without parentheses of course).

8.1.6 The *vapply()* function

The *vapply()* function is functionally the same as the *sapply()* function. You can use it to squeeze some more speed out of your code, by telling R what sort of stuff your function will return. When using the *vapply()* function you are forced to think about what your function is going to return and, therefore, it can also be safer to use. To remember the functionality, think of the letter "v" in front, as meaning verify (it checks your expected output). As an example consider:

```
cities <- c("New York", "London", "Cape Town")
# tell vapply you are expecting 1 numeric value for each element
vapply(X = cities, FUN = nchar, FUN.VALUE = numeric(1))
```

```
#>  New York    London Cape Town
#>         8         6         9
```

If you would have specified `FUN.VALUE = numeric(2)`, R will return an error message, because you said you expected 2 numeric values and it provided only 1.

One more example of *vapply()* using an *anonymous* function:

```
# create a list with data
dat <- list(a = 1:10, b = 11:20)
# return minimum and maximum for each element of the list
vapply(X = dat,
       FUN = function(x) {return(c(min = min(x), max = max(x)))},
       FUN.VALUE = c(min = 0, max = 0))
```

```
#>      a  b
#> min  1 11
#> max 10 20
```

8.1.7 Exercise SWIRL: *vapply()* and *tapply()*

Start *swirl()* from the package **swirl**, as you have done before, and work through lesson 11. In this lesson, you'll learn how to use *vapply()* and *tapply()*, two other members of R's **apply()* family of functions.

8.1.8 Exercise: *tapply()* application on the *chickwts* data set

In this exercise you will use the ***chickwts*** data frame from the R base **datasets** (check the help page for this data set). You probably have noticed by now, that R comes default with a lot of built-in data sets The **datasets** package is one of them, but often package developers provide data sets with their package for testing purposes and example use of the functions contained in their package as well.

The ***chickwts*** data frame contains data from an experiment to measure and compare the effectiveness of various feed supplements on the growth rate of chickens. "Eyeball" the data yourself to get a feeling of how the data frame is structured (number of observations, variable names, missing values, *etc.*).

a. With the *table()* function you can easily find the number of chicks per feed (try it yourself, for use of the *table()* function see section Section 3.1.3). Now try to obtain the same answer using the *tapply()* function and assign the outcome to an R object named ***nChicks*** (number of chicks per feed). [Hint: How would you determine the number of observations in a vector?]

b. Using the *tapply()* function also calculate the mean and standard deviation of the body weight of chicks per feed, assign them, respectively, to R objects ***meanChicks*** and ***sdChicks***.

c. To calculate 90% Confidence Intervals for the mean body weight of chicks per feed, you would need to find six Student t-values with $\alpha = 0.10$. In this lesson vectorization of a function was mentioned (function taking multiple input values). Using the ***nChicks*** object obtain the six required t-values needed to calculate the 90% Confidence Intervals for the mean body weight of chicks per feed. [Hint: Have you used the correct degrees of freedom?]

d. Create an object ***semChicks***, which stores the standard error of the mean body weight of chicks per feed. Finally calculate the ***lbound*** and ***ubound*** objects, representing the lower and upper boundaries of the 90% Confidence Interval for the mean body weight of chicks per feed using the objects you have created in all the questions above.

Try the following in a code chunk of your Rmarkdown report to generate a nicely formatted table:

```
knitr::kable(x = rbind(mean = meanChicks, # row with means
                       sd = sdChicks,     # row with standard deviations
                       n = nChicks,       # row with number of observations
                       sem = semChicks,   # row with standard errors of the mean
                       t = tvalues,       # t-values at 90% CI and appropriate df
                       lower = lbound,    # lower boundaries of 90% CI
                       upper = ubound),   # upper boundaries of 90% CI
```

```
            digits = 3,                  # number of digits to display in table
            caption = "\\label{tbl:ci_chickweights}90% CI for chickweights per feed.")
```

A reference to this table can be made in your Rmarkdown report by using `\ref{tbl:ci_chickweights}` in the Rmarkdown text of your report, e.g. somewhere in your Rmd-file write:
`"see Table \ref{tbl:ci_chickweights}"`.

8.1.9 The *mapply()* function

mapply() is a multivariate apply of sorts, which applies a function in parallel over a set of arguments. The function is multivariate in the sense that your function must accept multiple arguments and *mapply()* applies the provided function (the first argument supplied to *mapply()*) to each element. As output the *mapply()* tries to return a vector.

```
str(mapply)
```

```
#> function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

where:

- `FUN` is a function to apply,
- `...` arguments to vectorize over,
- `MoreArgs` is a list of other arguments to `FUN`,
- `SIMPLIFY` a logical which indicates whether the result should be simplified.

As an example consider the following command, which is tedious to type:

```
list(rep(x = 1, times = 4),
     rep(x = 2, times = 3),
     rep(x = 3, times = 2),
     rep(x = 4, times = 1))
```

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

The same can be achieved with *mapply()* in a much shorter way:

```
mapply(FUN = rep, 1:4, 4:1)
```

If for example you want to generate a list of drawings from a normal distribution, where you want 1 value from $N(\mu = 1, \sigma = 2)$, 2 values from $N(\mu = 2, \sigma = 2)$, ..., 5 values from $N(\mu = 5, \sigma = 2)$. In R terms, this would look like:

```
# create a function to generate noise (drawings from a normal distribution)
noise <- function(n, mean, sd) {
  rnorm(n, mean, sd)
}
set.seed(seed = 1492)
list(noise(n = 1, mean = 1, sd = 2),
     noise(n = 2, mean = 2, sd = 2),
     noise(n = 3, mean = 3, sd = 2),
     noise(n = 4, mean = 4, sd = 2),
     noise(n = 5, mean = 5, sd = 2))
```

```
#> [[1]]
#> [1] -0.1796952
#>
#> [[2]]
#> [1]  0.2026851 -1.2482220
#>
#> [[3]]
#> [1] 5.095843 2.443423 3.724986
#>
#> [[4]]
#> [1] 2.280275 4.108430 1.683586 4.829502
#>
#> [[5]]
#> [1] 5.699768 2.771767 2.322856 5.425828 4.207406
```

You could try vectorizing the *noise()* function. Given our example you could try:

```
# vectorize the function
set.seed(seed = 1492)
noise(n = 1:5, mean = 1:5, sd = 2)
```

```
#> [1] -0.1796952  0.2026851 -0.2482220  6.0958431  4.4434227
```

Here you see, that the result is not what you wanted. Vectorization of the *noise()* function resulted in a single drawing from $N(\mu, \sigma = 2)$ with μ increasing from 1 to 5. The function call is equivalent to `noise(n = 5, mean = 1:5, sd = 2)` (check it yourself, but make sure you use the same pseudo random numbers, e.g. `set.seed(1492)`).

The *mapply()* function can be used for instant vectorization to obtain the desired list. The R code how to get the desired list:

```
set.seed(seed = 1492)
mapply(FUN = noise, n = 1:5, mean = 1:5, sd = 2)
```

```
#> [[1]]
#> [1] -0.1796952
#>
#> [[2]]
#> [1]  0.2026851 -1.2482220
#>
#> [[3]]
#> [1] 5.095843 2.443423 3.724986
#>
#> [[4]]
#> [1] 2.280275 4.108430 1.683586 4.829502
#>
#> [[5]]
#> [1] 5.699768 2.771767 2.322856 5.425828 4.207406
```

You see, that now the result is equivalent to the original list generated with the *list()* function above, just much shorter code-wise.

8.1.10 The *split()* function

The *split()* function takes a vector or other objects and splits it into groups determined by a factor or list of factors.

```
str(split)
```

```
#> function (x, f, drop = FALSE, ...)
```

where,

- `x` is a vector (or list) or data frame,

- `f` is a factor (or coerced to one) or a list of factors,
- `drop` is a logical indicating if levels that do not occur should be dropped.

For example:

```
set.seed(seed = 1492)
# create a vector
x <- c(rnorm(n = 10), runif(n = 10), rnorm(n = 10, mean = 1))
# create a factor, levels 1,2,3 each level 10 times
f <- gl(n = 3, k = 10)
# split the vector x based on the factor f
split(x = x, f = f)
```

```
#> $`1`
#>  [1] -0.58984761 -0.89865744 -1.62411102  1.04792156 -0.27828867  0.36249296
#>  [7] -0.85986275  0.05421515 -1.15820705  0.41475084
#>
#> $`2`
#>  [1] 0.63678712 0.38116245 0.13261463 0.59908582 0.09035503 0.93201983
#>  [7] 0.58430305 0.56182740 0.34594304 0.97138688
#>
#> $`3`
#>  [1] -0.04490487  2.01851980  3.14282620  0.80095461 -0.92561405  1.66754002
#>  [7]  0.02921208  0.85405470  2.05158009 -0.22787230
```

A common idiom is *split()* followed by an *lapply()* or *sapply()*, *e.g.* :

```
lapply(X = split(x = x, f = f), FUN = mean)
```

```
#> $`1`
#> [1] -0.3529594
#>
#> $`2`
#> [1] 0.5235485
#>
#> $`3`
#> [1] 0.9366296
```

```
# or
sapply(X = split(x = x, f = f), FUN = mean)
```

```
#>          1          2          3
#> -0.3529594  0.5235485  0.9366296
```

The first example showed *split()* in combination with *lapply()* and *sapply()* for a vector, but, as stated, can also be used on a data frame, *e.g.* the ***airquality*** data frame from the base **datasets** package:

```
class(airquality)
```

```
#> [1] "data.frame"
```

```
head(airquality)
```

```
#>   Ozone Solar.R Wind Temp Month Day
#> 1    41     190  7.4   67     5   1
#> 2    36     118  8.0   72     5   2
#> 3    12     149 12.6   74     5   3
#> 4    18     313 11.5   62     5   4
#> 5    NA      NA 14.3   56     5   5
#> 6    28      NA 14.9   66     5   6
```

```
# split airquality data frame by month
splitAirquality <- split(x = airquality, f = airquality$Month)
```

```
sapply(X = splitAirquality,
       FUN = function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE))
```

```
#>                 5         6          7          8         9
#> Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
#> Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
#> Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

```
# can also be combined in one function call
# sapply(X = split(x = airquality, f = airquality$Month),
#        FUN = function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE))
```

8.1.11 Exercise: Application of *split()* and *lapply()* / *sapply()* on the *iris* data set

In this exercise you will again use the ***iris*** data set introduced in exercise Section 8.1.2.

What is the mean of sepal length and width, and petal length and width for the species *Iris virginica*? This can be achieved with as little as two function calls. [Hint: To answer this question you will have to use splitting, subsetting, and either the *lapply()* or *sapply()* function]

8.1.12 Exercise Loop Functions and Regression: Repeated Measurements of Rat Weights

In this exercise you will look at repeated measurements data. You have repeated measurements if multiple observations are made on the same experimental unit, usually over time. In this exercise you are going to look at data on rat growth. Three treatments (a control and two chemicals added to drinking water) were applied to 10, 7, and 10 rats, respectively. The assignment of treatments over rats was done randomly. Furthermore, each rat was treated individually. Therefore, the rats are the experimental units. The weights of the rats were recorded at the start of the experiment and after 1, 2, 3, and 4 weeks of treatment. The data used here originate from an article by Box (1950) [17].

- Read the data file `ratgrowth.csv` into R.
- Make sure that the variable named `Trt` is a factor.
- Give the following names to the levels of this factor: "control", "thyroxin", "thiouracil", using the function *levels()*.

The first question, that needs to be answered, is whether the treatment has an effect on the final weight after 4 weeks.

- Make a graphical summary of final weight after 4 weeks using side-by-side boxplots.
- Perform one-way ANOVA, explaining the weight after 4 weeks from the treatment.
- Give the ANOVA table.
- Make pairwise comparisons between the group means using an appropriate method. Which groups are significantly different?

To study the effect of time on weight, you may take time as a second factor and perform two-way ANOVA. However, this approach is not valid, because of the repeated measurements per rat. The two-way ANOVA assumes that all observations are independent, which will not be the case for the repeated measurements on the same rat: if a rat is relatively heavy on e.g. day 1, it will also be relatively heavy on day 2 and later on. Repeated measurements tend to be correlated. Therefore, the problem is dealt with in the following way. For each individual rat you calculate the regression of weight on time and store the slope. For a fast growing animal the slope will be steeper. Next, compare the slopes between the different treatment groups.

First program a `for` loop in which for each rat the regression of weight on time is performed. Take the following steps:

- Define the index i for the loop, and construct the `for` loop.
- Select the data for rat i from the rat data frame
- Regress the weights of rat i on the corresponding time points (times are 0, 1, 2, 3, 4).

- Save the slopes from the regression in a vector.
- Make a new data frame with two columns, consisting of vector `Trt` and the vector with slopes.
- Perform one-way ANOVA for the slopes to study whether there are differences in growth rate between the treatment groups.

Instead of programming a `for` loop, functions from the **apply()* family can also be used.

- Store the five columns containing the weights of the rats in a separate data matrix.
- As a starter, use function *apply()* to obtain per rat the standard deviation of weight over the five time points. Letting the function *apply()* loop through the rows of the matrix you created in the previous question.
- Use *apply()* to obtain the slopes per rat. For this to work, define a small function that regresses a dependent vector (of length 5) on a regressor with the values 0, 1, 2, 3, 4 (which are the week numbers), and which returns the slope from the simple linear regressions.
- Again, make a new data frame with two columns, consisting of vector `Trt` and the vector with slopes.
- Again, perform one-way ANOVA for the slopes to study whether there are differences in growth rate between the treatment groups.

8.2 Command Reference

Table 8.1: Lesson 8 commands

Command	Description
apply()	apply functions over array margins
colMeans()	form column means
colSums()	form column sums
gl()	generate factor levels
lapply()	apply a function over a list or vector
mapply()	apply a function to multiple list or vector arguments
max()	returns the maximum of the input values
min()	returns the minimum of the input values
nchar()	count the number of characters(or bytes or width)
rowMeans()	form row means
rowSums()	form row sums
runif()	generate random numbers from a continuous uniform distribution
sapply()	same as *lapply()* with possible simplification
split()	divide the data into the groups defined by a factor
t()	transpose a matrix or data frame
tapply()	apply a function over a ragged array
unlist()	flatten lists

Lesson 9

R Base Graphics, Non-Parametric Tests, Fisher Exact Test and χ^2-Tests

9.1 R Base Graphics

An important reason, why statisticians and data analysts use R, is the excellent ability to produce a wide range of graphics to quickly and easily visualize data. There are multiple reasons, why R users want to use graphical representations (graphs or plots) in data analysis:

1. To understand data properties
2. To find patterns in data
3. To suggest modeling strategies
4. To "debug" analyses
5. To communicate results

The first four are summarizing, what is usually called "Exploratory Data Analysis".

To make graphs R provides several plotting systems:

- Base R [18], provided by the base packages:
  - **graphics** contains plotting functions for the "base" graphing system, including *plot()*, *hist()*, *boxplot()* and many others.
  - **grDevices** contains all the code implementing the various graphics devices, including *X11()*, *pdf()*, *postscript()*, *png()*, *jpeg()*, *etc.*
- Lattice [19], provided by the packages:
  - **lattice** contains code for producing *Trellis* graphics, which are independent of the base graphics system; includes, among others, functions like *xyplot()*, *bwplot()*, *levelplot()*.
  - **grid** implements a different graphing system independent of the base system. The **lattice** package builds on top of **grid** and functions, from this package, are seldom called directly.
  - **latticeExtra** contains extra graphical utilities based on the **lattice** package
- Ggplot2 [20] or its successor Ggvis [21], provided, respectively, by the packages:
  - **ggplot2**, which forms a system for 'declaratively' creating graphics, based on "The Grammar of Graphics". You provide the data, and tell **ggplot2** how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.
  - **ggvis**, which forms an implementation of an interactive grammar of graphics, by taking the best parts of **ggplot2**, combining it with the reactive framework of **shiny** and drawing web graphics using the package **vega**.

In this lesson the Base R plotting system will be introduced, which is the oldest graphics system in R. The Base R plotting system is based on the, so-called, "Artist's palette" model. In this model you start with a blank canvas, usually with a *plot()* function (or similar) and build up from there by adding annotation functions, to add to or modify (text, lines, points, axis) an existing graph/plot.

The Base R plotting system conveniently mirrors, how data analysts think of building plots and analyzing data. It has, however, one major drawback, that you can not go back once a plot has been started, e.g. to adjust margins. You need to plan in advance, or keep track of how you build up your plot in a script, which you can modify and rerun whenever you want.

Graphs/plots created with the base R graphics system are difficult to "translate" to others once a new plot has been created (there is no grammar of graphics, i.e. no graphical "language"). Creation of graphs or plots is just a series of R commands, but this also means that it provides an easy and fast system.

9.1.1 Initializing a new plot with high-level plotting functions

As mentioned above, the first phase in creating a base plot is initializing a new plot. This is done with, what is called in the "Introduction to R" manual [22], a high-level command. High-level plotting functions are designed to create a complete plot of the data passed as arguments to the function. When appropriate, axis, labels, and titles are automatically generated, unless you request otherwise. Keep in mind that high-level plotting commands **always** start a new plot, erasing the current plot if necessary.

9.1.1.1 The *plot()* function

The *plot()* function is one of the most used high-level plotting functions in R. It is a generic function. Meaning that the type of plot created depends on the type or class of the first argument.

```
str(plot)
```

```
#> function (x, y, ...)
```

If `x` and `y`, are both vectors, *plot()* creates a scatterplot of `y` versus `x` (the same is achieved by `plot(y ~ x)`. When `x` is passed as a factor and `y` is passed as a numeric vector, the *plot()* function creates boxplots of `y` for each level of `x`.

```
# set parameters for 2 plots side-by-side
par(mfrow = c(1, 2),
    mar = c(4, 4, 3, 0.1),
    mgp = c(2,0.75,0),
    cex = 0.8)
iris <- datasets::iris
class(iris$Sepal.Length)
```

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

```
class(iris$Petal.Width)
```

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

```
plot(x = iris$Sepal.Length, y = iris$Petal.Width)
class(iris$Species)
```

```
#> [1] "factor"
```

```
plot(x = iris$Species, y = iris$Sepal.Length)
```

The scatter- and boxplot, of the code displayed above, are shown in Figure 9.1

In case only the `x` argument is passed as a time series, this produces a time-series plot. When the `x` argument is a factor, a barplot of the factor is created.

```
# set parameters for 2 plots side-by-side
par(mfrow = c(1, 2),
    mar = c(4, 4, 3, 0.1),
    mgp = c(2,0.75,0),
    cex = 0.8)
ldeaths <- datasets::ldeaths # time series data set giving the monthly deaths from
                             # bronchitis, emphysema and asthma in the UK, 1974-1979
class(ldeaths)
```

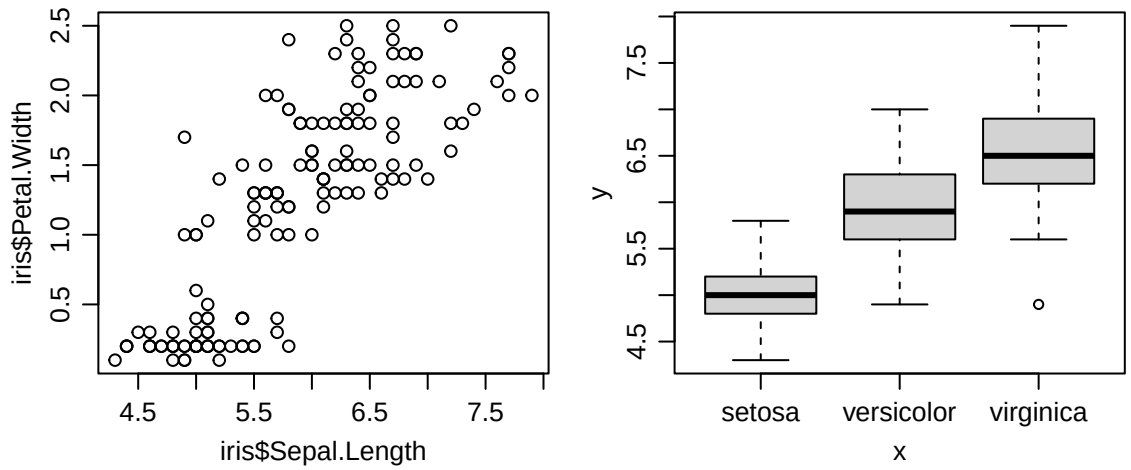


Figure 9.1: Examples of a scatterplot and side-by-side boxplots generated with the *plot()* function

```
#> [1] "ts"
```

```
plot(ldeaths)
plot(x = iris$Species)
```

Examples of both the time series plot and the barplot, as given in the code printed above, are shown in Figure 9.2.

9.1.1.2 Other high-level plotting functions

The **graphics** package contains a load of other high-level functions to produce different types of plots. Some of these functions, really worth mentioning, are (check out their individual help pages with `?functionname` for specification of function arguments and examples):

```
qqnorm(x)
qqline(x)
qqplot(x, y)
```

Distribution-comparison plots. The first, `qqnorm(x)`, plots the numeric vector ***x*** against the expected Normal order scores (a normal scores plot) and the second, `qqline(x)`, adds a straight line to such a plot by drawing a line through the distribution and data quartiles. Officially the ***qqline()*** function is a low-level plotting command, because it adds to the existing ***qqnorm()*** plot. However, in this case the one can not exist without the other in generating a Q-Q plot, and, therefore, its is usually counted as a high-level plotting command. The third form, `qqplot(x, y)`, plots the quantiles of ***x*** against those of ***y*** to compare their respective distributions.

```
hist(x)
```

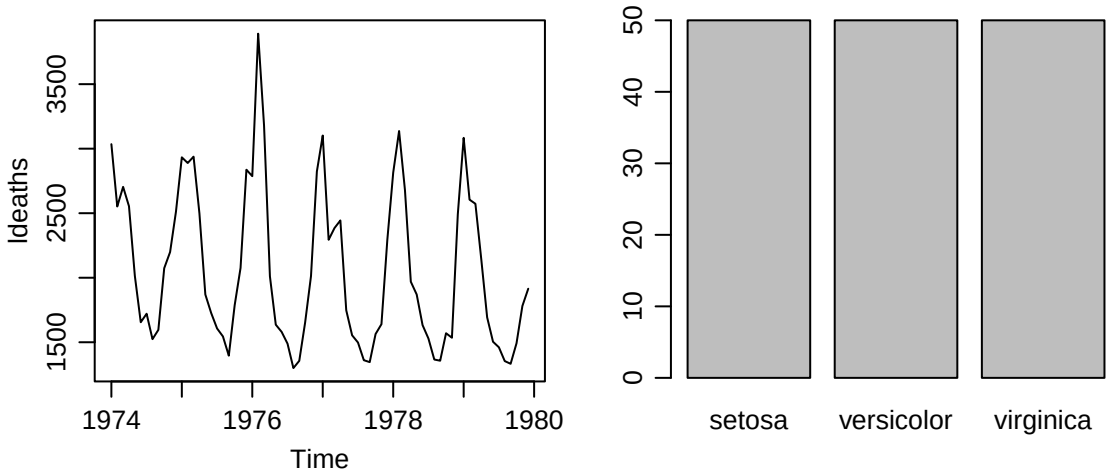


Figure 9.2: Examples of a time series plot and barplot generated with the *plot()* function

```
hist(x, nclass = n)
hist(x, breaks = b, ...)
```

Produces a histogram of the numeric vector x. A sensible number of classes is usually chosen, but a recommendation can be given with the `nclass` argument. Alternatively, the breakpoints can be specified exactly with the `breaks` argument. If the `probability = TRUE` argument is given, the bars represent relative frequencies divided by bin width instead of counts.

```
boxplot(x, ...)
```

Produce box-and-whisker plot(s) of the given (grouped) values. The generic function *boxplot()* currently has a default method:

```
boxplot(iris$Sepal.Length)
```

and a formula interface:

```
boxplot(iris$Sepal.Length ~ iris$Species)
```

If multiple groups are supplied either as multiple arguments or via a formula, parallel boxplots will be plotted, in the order of the arguments or the order of the levels of the factor (see `?factor`). Missing values are ignored when forming boxplots.

```
barplot(height, ...)
```

Creates a bar plot with vertical or horizontal bars. For example, as shown in Figure 9.3:

```
VADeaths <- datasets::VADeaths
barplot(height = VADeaths,
        legend.text = rownames(VADeaths),
        args.legend = list(x = "topright", bty = "n"),
        main = "Death rates per 1000 in Virginia in 1940.")
```

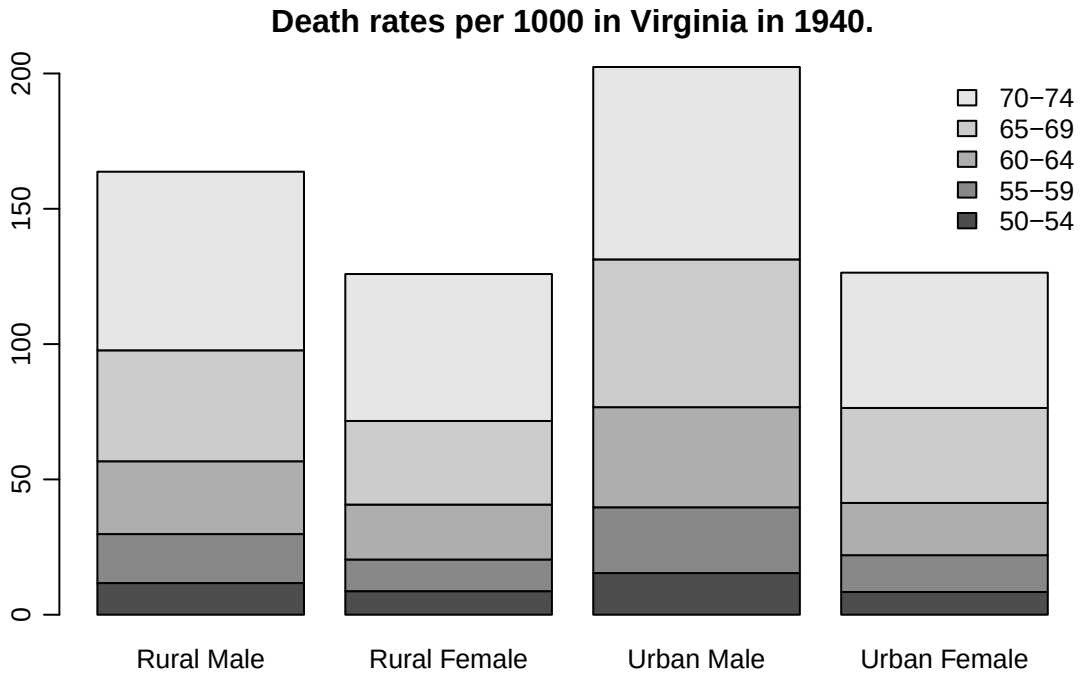


Figure 9.3: Example of a stacked barplot generated with the *barplot()* function

```
image(x, y, z, ...)
contour(x, y, z, ...)
persp(x, y, z, ...)
```

Plots of three variables. The *image()* plot draws a grid of rectangles using different colors to represent the value of z, the *contour()* plot draws contour lines to represent the value of z, and the *persp()* plot draws a 3D surface.

```
matplot(x, y, ...)
```

Plot the columns of one matrix against the columns of another. Example of what you can do with the *matplot()* function is shown in Figure 9.4. The code for this example you can find in the examples section on the help page of the *matplot()* function.

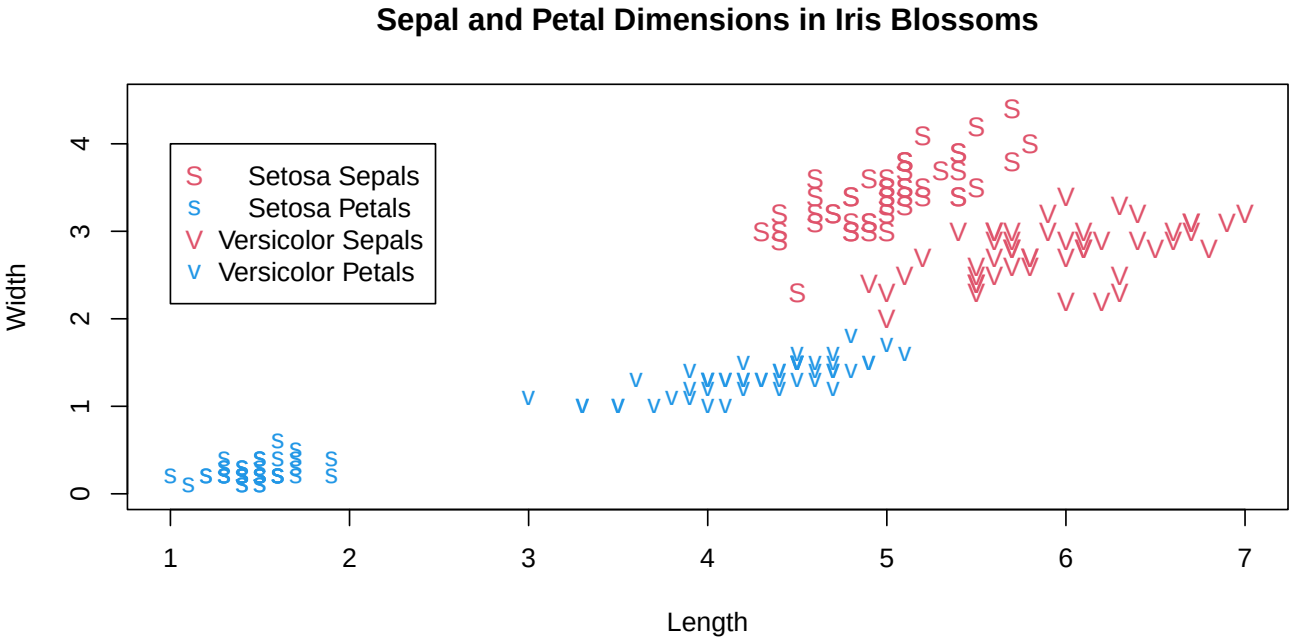


Figure 9.4: Example of the *matplot()* function

```
pairs(x, ...)
```

As you have seen in the exercise about multiple regression (Section 7.2.1), the *pairs()* function produces a matrix of scatterplots. It is one of two very useful functions in R for representing multivariate data.

9.1.1.3 Arguments to high-level plotting functions

There are a number of arguments which may be passed to high-level graphics functions in order to generate a complete plot in one function call. These arguments are defined as:

```
add = TRUE
```

Forces the function to act as a low-level graphics function, superimposing the plot on the current plot (some functions only).

```
axes = FALSE
```

Suppresses generation of axes; useful for adding your own custom axes with the *axis()* function. The default, `axes = TRUE`, means include axes.

```
log = "x"
log = "y"
log = "xy"
```

Causes the *x*, *y* or both axes to be logarithmic. This will work for many, but not all, types of plot.

The `type` argument controls the type of plot produced (see Figure 9.5 for examples), as follows:

```
type = "p"
```

Plot individual points (the default)

```
type = "l"
```

Plot lines

```
type = "b"
```

Plot points connected by lines (both).

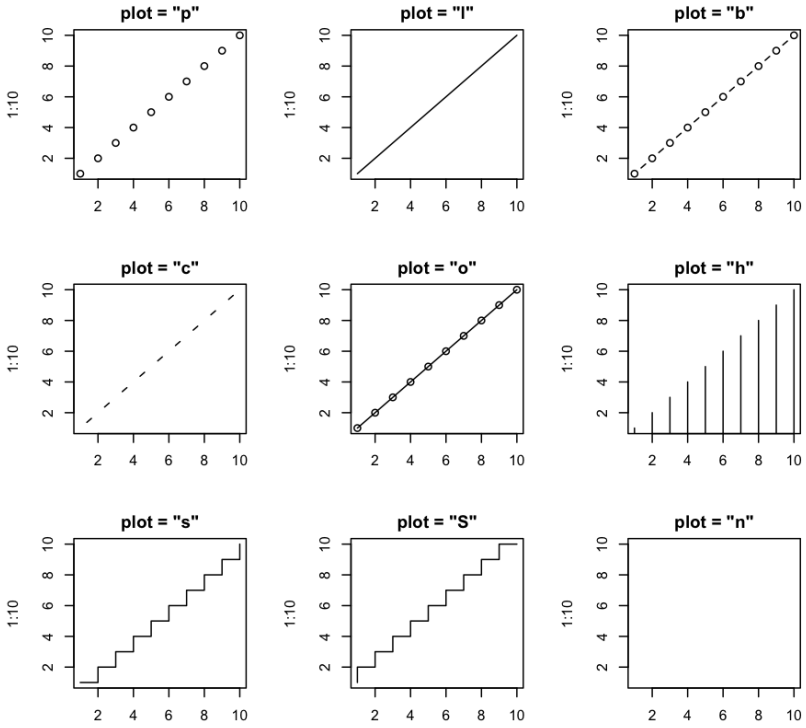


Figure 9.5: Plot types as specified with the `type` argument.

`type = "o"`

Plot points overlaid by lines.

`type = "h"`

Plot vertical lines from points to the zero axis (high-density).

```
type = "s"
type = "S"
```

Step-function plots. In the first form, `type = "s"`, the top of the vertical defines the point; in the second, `type = "S"`, the bottom.

`type = "n"`

No plotting at all. However, axes are still drawn (by default) and the coordinate system is set up according to the data. Ideal for creating plots with subsequent low-level graphics functions.

```
xlab = "text string"
ylab = "text string"
```

Axis labels for the x and y axes. Use these arguments to change the default labels, usually the names of the objects used in the call to the high-level plotting function.

```
xlim = c(xmin, xmax)
ylim = c(ymin, ymax)
```

Coordinate ranges for the x and y axes in a plot. This will work for many, but not all, high-level plotting functions. By default, the range of the data supplied to the high-level plotting function is used, unless explicitly set with the `xlim` and `ylim` arguments.

`main = "text string"`

Figure title, placed at the top of the plot in a large font.

`sub = "text string"`

Sub-title, placed just below the x-axis in a smaller font.

9.1.2 Adding to or modifying an initialized plot

Sometimes the high-level plotting functions don't produce exactly the kind of desired plot. In this case, low-level plotting commands can be used to add extra information (such as points, lines or text) to the current plot (created with a high-level plotting function).

Some of the more useful low-level plotting functions are:

```
points(x, y)
lines(x, y)
```

Adds points or connected lines to the current plot. *plot()*'s `type` argument can also be passed to these functions (and defaults to `"p"` for *points()* and `"l"` for *lines()*).

```
text(x, y, labels, ...)
```

Add text to a plot at points given by ***x***, ***y***. Normally labels is an integer or character vector in which case `labels[i]` is plotted at point `(x[i], y[i])`. The default is `1:length(x)`.

> Note: *text()* combined with *plot()*
>
> The *text()* function is often used in the sequence `plot(x, y, type = "n"); text(x, y, names)`.
>
> The *plot()* function argument `type = "n"` suppresses the points but sets up the axes, and the *text()* function supplies special characters, as specified by the character vector names for the points.

```
abline(a, b)
abline(h = y)
abline(v = x)
abline(lmObj)
```

Adds a line of slope b and intercept a to the current plot. `h = y` may be used to specify *y*-coordinates for the heights of horizontal lines to go across a plot, and `v = x` similarly for the *x*-coordinates for vertical lines. Also, ***lmObj*** may be list with a coefficients component of length 2 (such as the result of model-fitting functions), which are taken as an intercept and slope, in that particular order.

```
polygon(x, y, ...)
```

Draws a polygon defined by the ordered vertices in `(x, y)` and (optionally) shade it in with hatch lines, or fill it if the graphics device allows the filling of figures.

```
legend(x, y, legend, ...)
```

Adds a legend to the current plot at the specified position. Plotting characters, line styles, colors etc., are identified with the labels in the character vector legend. At least one other argument `v` (a vector the same length as legend) with the corresponding values of the plotting unit must also be given, as follows:

```
legend( , fill = v)
```

Colors for filled boxes

```
legend( , col = v)
```

Colors in which points or lines will be drawn

```
legend( , lty = v)
```

Line styles

```
legend( , lwd = v)
```

Line widths

```
legend( , pch = v)
```

Plotting characters (character vector)

```
title(main, sub)
```

Adds a title `main` to the top of the current plot in a large font and (optionally) a sub-title `sub` at the bottom in a smaller font.

`axis(side, ...)`

Adds an axis to the current plot on the side given by the first argument (1 to 4, counting clockwise from the bottom.) Other arguments control the positioning of the axis within or beside the plot, and tick positions and labels. Useful for adding custom axes after calling *plot()* with the `axes = FALSE` argument.

Low-level plotting functions usually require some positioning information (*e.g.*, x and y coordinates) to determine where to place the new plot elements. Coordinates are given in terms of *user coordinates* which are defined by the previous high-level graphics command and are chosen based on the supplied data.

Where `x` and `y` arguments are required, it is also sufficient to supply a single argument being a list with elements named `x` and `y`. Similarly, a matrix with two columns is also valid input.

9.1.2.1 Mathematical annotation

In some cases, it is useful to add mathematical symbols and formulae to a plot. This can be achieved in R by specifying an expression rather than a character string in any one of the *text()*, *mtext()*, *axis()*, or *title()* functions. For example, the following code draws the formula for the Binomial probability function:

```
text(x, y, expression(paste(bgroup("(", atop(n, x), ")"), p^x, q^{n-x})))
```

More information, including a full listing of the features available can obtained from within R using the commands:

```
help(plotmath)
example(plotmath)
demo(plotmath)
```

9.1.3 Using graphics parameters

When creating graphics, particularly for presentation or publication purposes, R's defaults do not always produce exactly that which is required. You can, however, customize almost every aspect of the display using graphics parameters. R maintains a list of a large number of graphics parameters which control things such as line style, colors, figure arrangement and text justification among many others. Every graphics parameter has a name (such as '`col`', which controls colors,) and a value (a color number, for example.)

A separate list of graphics parameters is maintained for each active device, and each device has a default set of parameters when initialized. Graphics parameters can be set in two ways: either permanently, affecting all graphics functions which access the current device; or temporarily, affecting only a single graphics function call.

9.1.3.1 Permanent changes: The *par()* function}

The *par()* function is used to access and modify the list of graphics parameters for the current graphics device.

`par()`

Without arguments, returns a list of all graphics parameters and their values for the current device. '`par(c("col", "lty"))` With a character vector argument, returns only the named graphics parameters (again, as a list).

`par(col = 4, lty = 2)`

With named arguments (or a single list argument), sets the values of the named graphics parameters, and returns the original values of the parameters as a list.

Setting graphics parameters with the *par()* function changes the value of the parameters permanently, in the sense that all future calls to graphics functions (on the current device) will be affected by the new value. You can think of setting graphics parameters in this way as setting "default" values for the parameters, which will be used by all graphics functions unless an alternative value is given.

> **Note: Global effect of the *par()* function**
>
> Calls to *par()* always affect the global values of graphics parameters, even when *par()* is called from within a function.

Changing graphics parameters globally is often undesirable behavior. Usually we want to set some graphics parameters, do some plotting, and then restore the original values so as not to affect the user's R session. You can restore the initial values by saving the result of *par()* when making changes, and restoring the initial values when plotting is complete.

```
oldpar <- par(col = 4, lty = 2)
<some plotting commands>
par(oldpar)
```

To save and restore all settable graphical parameters use:

```
oldpar <- par(no.readonly = TRUE)
  <plotting commands>
par(oldpar)
```

9.1.3.2 Temporary changes: Arguments to graphics functions

Graphics parameters may also be passed to (almost) any graphics function as named arguments. This has the same effect as passing the arguments to the *par()* function, except that the changes only last for the duration of the function call. For example: `plot(x, y, pch = "+")`, produces a scatterplot using a plus sign as the plotting character, without changing the default plotting character for future plots.

Unfortunately, this is not implemented entirely consistently and it is sometimes necessary to set and reset graphics parameters using the *par()* function.

9.1.4 Graphics parameters list

The following sections detail many of the commonly-used graphical parameters. The R help documentation for the *par()* function provides a more concise summary; this is provided as a somewhat more detailed alternative.

Graphics parameters will be presented in the following form:

`name = value`

A description of the parameter's effect. `name` is the name of the parameter, that is, the argument name to use in calls to *par()* or a graphics function. `value` is a typical value you might use when setting the parameter.

Note that axes is not a graphics parameter but an argument to a few `plot` methods: see `xaxt` and `yaxt`.

9.1.4.1 Graphical elements

R plots are made up of points, lines, text and polygons (filled regions). Graphical parameters exist which control how these *graphical elements* are drawn, as follows:

`pch = "+"`

Character to be used for plotting points. The default varies with graphics drivers, but it is usually a "○". Plotted points tend to appear slightly above or below the appropriate position unless you use "." as the plotting character, which produces centered points.

`pch = 4`

When `pch` is given as an integer between 0 and 25 inclusive, a specialized plotting symbol is produced. The symbols are shown in Figure 9.6.

Those from 21 to 25 may appear to duplicate earlier symbols, but can be colored in different ways: see the help on the *points()* function and its examples.

In addition, `pch` can be a character or a number in the range 32:255 representing a character in the current font.

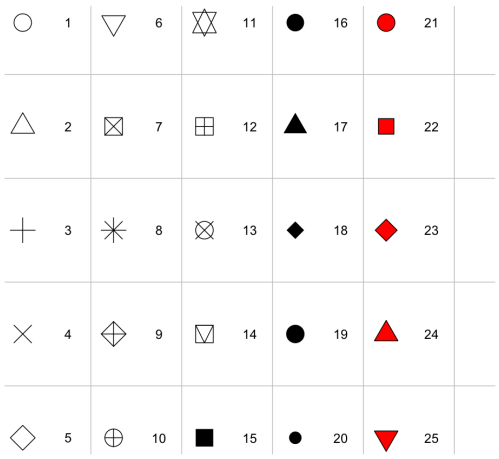


Figure 9.6: `pch` symbols.

`lty = 2`

Line types. Alternative line styles are not supported on all graphics devices (and vary on those that do) but line type 1 is always a solid line, line type 0 is always invisible, and line types 2 and onwards are dotted or dashed lines, or some combination of both. Figure 9.7 shows some possible values for `lty`.

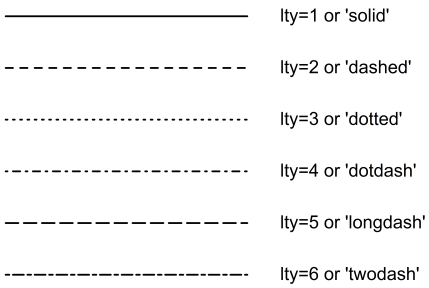


Figure 9.7: Line type examples as specified by `lty`.

`lwd = 2`

Line widths. Desired width of lines, in multiples of the "standard" line width. Affects axis lines as well as lines drawn with *lines()*, *etc.* Not all devices support this, and some have restrictions on the widths that can be used.

`col = 2`

Colors to be used for points, lines, text, filled regions and images. A number from the current palette (see `?palette` for help) or a named color. To see a list of possible color names execute *colors()* in the console or search in Google on "named colors r".

`col.axis`
`col.lab`
`col.main`
`col.sub`

The color to be used for axis annotation, x and y labels, main and sub-titles, respectively.

`font = 2`

An integer which specifies which font to use for text. If possible, device drivers arrange so that 1 corresponds to plain text, 2 to bold face, 3 to italic, 4 to bold italic and 5 to a symbol font (which include Greek letters).

`font.axis`
`font.lab`

`font.main`
`font.sub`

The font to be used for axis annotation, *x* and *y* labels, main and sub-titles, respectively.

`adj = -0.1`

Justification of text relative to the plotting position. 0 means left justify, 1 means right justify and 0.5 means to center horizontally about the plotting position. The actual value is the proportion of text that appears to the left of the plotting position, so a value of -0.1 leaves a gap of 10% of the text width between the text and the plotting position.

`cex = 1.5`

Character expansion. The value is the desired size of text characters (including plotting characters) relative to the default text size.

`cex.axis`
`cex.lab`
`cex.main`
`cex.sub`

The character expansion to be used for axis annotation, *x* and *y* labels, main and sub-titles, respectively.

9.1.4.2 Axes and tick marks

Many of R's high-level plots have axes, and you can construct axes yourself with the low-level *axis()* graphics function. Axes have three main components: the *axis line* (line style controlled by the `lty` graphics function argument), the *tick marks* (which mark off unit divisions along the axis line) and the *tick labels* (which mark the units.) These components can be customized with the following graphics parameters.

`lab = c(5, 7, 12)`

The first two numbers are the desired number of tick intervals on the *x* and *y* axes respectively. The third number is the desired length of axis labels, in characters (including the decimal point.) Choosing a too-small value for this parameter may result in all tick labels being rounded to the same number!

`las = 1`

Orientation of axis labels. 0 means always parallel to axis, 1 means always horizontal, and 2 means always perpendicular to the axis.

`mgp = c(3, 1, 0)`

Positions of axis components. The first component is the distance from the axis label to the axis position, in text lines. The second component is the distance to the tick labels, and the final component is the distance from the axis position to the axis line (usually zero). Positive numbers measure outside the plot region, negative numbers inside.

`tck = 0.01`

Length of tick marks, as a fraction of the size of the plotting region. When `tck` is small (less than 0.5) the tick marks on the *x* and *y* axes are forced to be the same size. A value of 1 gives grid lines. Negative values give tick marks outside the plotting region. Use `tck = 0.01` and `mgp = c(1, -1.5, 0)` for internal tick marks.

`xaxs = "r"`
`yaxs = "i"`

Axis styles for the *x* and *y* axes, respectively. With styles `"i"` (internal) and `"r"` (the default) tick marks always fall within the range of the data, however style `"r"` leaves a small amount of space at the edges. (The S language has other styles not implemented in the R language.)

9.1.4.3 Figure margins

A single plot in R is known as a figure and comprises a *plot region* surrounded by margins (possibly containing axis labels, titles, *etc.*) and (usually) bounded by the axes themselves. A typical figure is shown in Figure 9.8.

Graphics parameters controlling figure layout include:

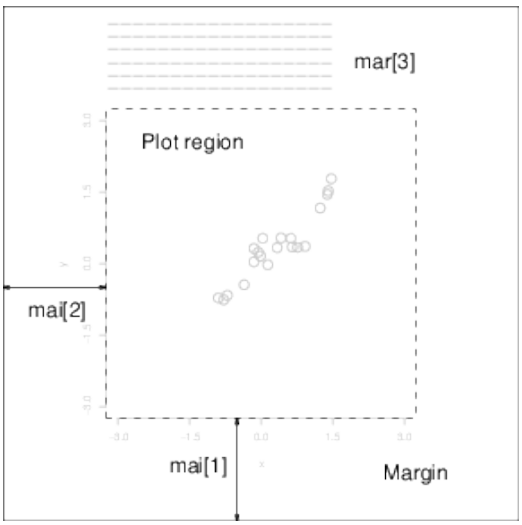


Figure 9.8: A typical figure with *plot region* and margins.

`mai = c(1, 0.5, 0.5, 0)`

Widths of the bottom, left, top and right margins, respectively, measured in inches.

`mar = c(4, 2, 2, 1)`

Similar to mai, except the measurement unit is text lines.

The function arguments `mar` and `mai` are equivalent in the sense that setting one changes the value of the other. The default values chosen for this parameter are often too large; the right-hand margin is rarely needed, and neither is the top margin if no title is being used. The bottom and left margins must be large enough to accommodate the axis and tick labels. Furthermore, the default is chosen without regard to the size of the device surface: for example, using the *postscript()* driver with the `height = 4` argument will result in a plot which is about 50% margin unless `mar` or `mai` are set explicitly. When multiple figures are in use (see below) the margins are reduced, however this may not be enough when many figures share the same page.

9.1.4.4 Multiple figure environment

R allows you to create an *n* by *m* array of figures on a single page. Each figure has its own margins, and the array of figures is optionally surrounded by an *outer margin*, as shown in Figure 9.9.

The graphical parameters relating to multiple figures are as follows:

```
mfcol = c(3, 2)
mfrow = c(2, 4)
```

Set the size of a multiple figure array. The first value is the number of rows; the second is the number of columns. The only difference between these two parameters is that setting mfcol causes figures to be filled by column; `mfrow` fills by rows.

The layout in the Figure could have been created by setting `mfrow = c(3, 2)`; the figure shows the page after four plots have been drawn.

Setting either of these can reduce the base size of symbols and text (controlled by `par("cex")` and the point size of the device). In a layout with exactly two rows and columns the base size is reduced by a factor of 0.83: if there are three or more of either rows or columns, the reduction factor is 0.66.

`mfg = c(2, 2, 3, 2)`

Position of the current figure in a multiple figure environment. The first two numbers are the row and column of the current figure; the last two are the number of rows and columns in the multiple figure array. Set this parameter to jump between figures in the array. You can even use different values for the last two numbers than the *true* values for unequally-sized figures on the same page.

`fig = c(4, 9, 1, 4) / 10`

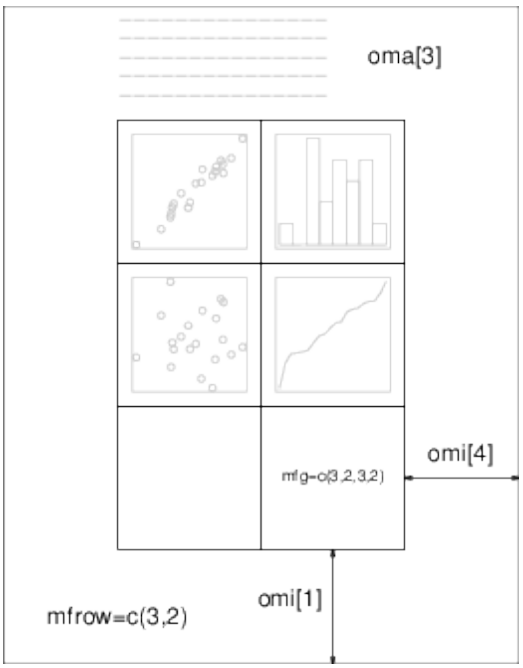


Figure 9.9: A multiple figure environment with outer margins.

Position of the current figure on the page. Values are the positions of the left, right, bottom and top edges respectively, as a percentage of the page measured from the bottom left corner. The example value would be for a figure in the bottom right of the page. Set this parameter for arbitrary positioning of figures within a page. If you want to add a figure to a current page, use `new = TRUE` as well (unlike S).

```
oma = c(2, 0, 3, 0)
omi = c(0, 0, 0.8, 0)
```

Size of outer margins. Like `mar` and `mai`, the first measures in text lines and the second in inches, starting with the bottom margin and working clockwise.

Outer margins are particularly useful for page-wise titles, *etc.* Text can be added to the outer margins with the *mtext()* function with argument `outer = TRUE`. There are no outer margins by default, however, so you must create them explicitly using `oma` or `omi`.

More complicated arrangements of multiple figures can be produced by the *split.screen()* and *layout()* functions, as well as by the **grid** and **lattice** packages.

9.1.5 Graphics Devices in R

R can generate graphics (of varying levels of quality) on almost any type of display or printing device. Before this can begin, however, R needs to be informed what type of device it is dealing with. This is done by starting a *device driver*. The purpose of a device driver is to convert graphical instructions from R ("draw a line," for example) into a form that the particular device can understand.

Device drivers are started by calling a device driver function. There is one such function for every device driver: type `help(Devices)` or `?Devices` for a list of them all. For example, issuing the command: *postscript()*, causes all future graphics output to be sent to the printer in PostScript format. Some commonly-used device drivers are:

`X11()`

For use with the X11 window system on Unix-alikes.

`windows()`

For use on Windows.

`quartz()`

For use on macOS.

`postscript()`

For printing on PostScript printers, or creating PostScript graphics files.

`pdf()`

Produces a PDF file, which can also be included into PDF files.

`png()`

Produces a bitmap PNG file (not always available: see its help page).

`jpeg()`

Produces a bitmap JPEG file, best used for image plots (not always available: see its help page).

When you have finished with a device, be sure to terminate the device driver by issuing the command:

```
dev.off()
```

This ensures that the device finishes cleanly; for example in the case of hard copy devices this ensures that every page is completed and has been sent to the printer. (This will happen automatically at the normal end of a session.)

In RStudio graphs are created directly on the Plots tab of Panel 4 (see Figure 1.5). RStudio opens a device for you. Once your graph is complete, you can select Export in the menu of the Plot tab and choose in which format you want to save it. If you want to create a graph in a script or function and save, you need to specify the device and close it in your script or function.

9.1.6 Exercise SWIRL: Base Graphics

Start *swirl()* from the package **swirl**, as you have done before, and work through lesson 15. In this lesson, you'll learn about the Base Graphics System in R.

9.1.7 Exercise: Creating your own Q-Q plot

In O&L Section 4.14 (in both the 6^{th} and 7^{th} Edition) you can read in detail how you can evaluate whether or not a population distribution is normal. You do this by constructing a Q-Q plot and performing a test of Normality (e.g. Shapiro-Wilk test or Kolmogorov-Smirnov test). There is also a knowledge clip available about the construction of Q-Q plots on YouTube (see https://youtu.be/vBZaRSj1Aic).

The steps involved in building a Q-Q plot are:

1. Ranking the observed values, with averaging for ties
2. Assigning proportions to the ranked observed values with a proportion estimation formula (e.g. Blom's)
3. Calculating the theoretical standard normal quantiles for proportion values of the ranked observed values
4. Transforming the calculated theoretical standard normal quantiles with the mean and standard deviation from the original data into expected normal values
5. Plotting the expected normal values against the observed values and adding the $y = x$ line

In this exercise you will use the data about the Fish quality and Delay problem, presented in Lesson 4, to build your own Q-Q plot by using only the residuals and performing a test of Normality on the residuals.

When using SPSS you would use from the top menu **Analyze** → **Descriptive Statistics** → **Explore**... to build a Q-Q plot and perform a test of Normality on the Unstandardized Residual [RES_1]. Figure 9.10 shows the Q-Q plot generated with SPSS.

You see that it differs from the Q-Q plot created in Lesson 4 with the functions *qqnorm()* and *qqline()*.

Write a script, that recreates in R the Normal Q-Q plot of the Unstandardized Residual as generated with SPSS (without the light grey color of the plot region). Use the following:

- The data about `quality` and `delay` as contained in the `fish_quality_vs_delay.csv` file
- The *rank()* function for ranking the residuals. Check the help page for function arguments!

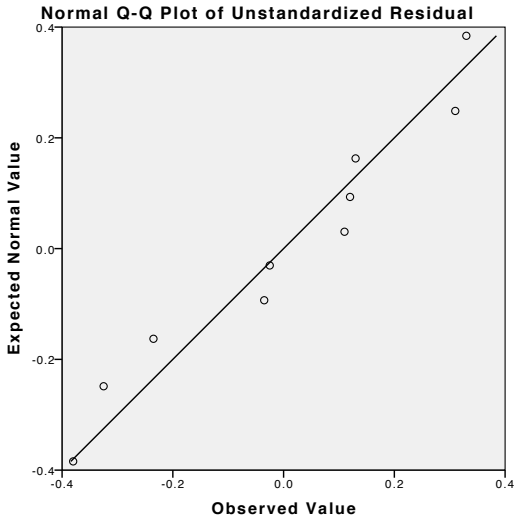


Figure 9.10: SPSS generated Normal Q-Q Plot of Unstandardized Residual for the Fish quality vs. Delay problem.

- Blom's proportion estimation formula, $p = \frac{i-0.375}{n+0.25}$ with i = *observation rank* and n = *total number of observations*, to calculate the proportions of the ranked residuals.
- The *qnorm()* function to calculate the theoretical standard normal quantiles (z-values) for proportion values of the ranked residuals. Read the help page of *qnorm()* for specification of function arguments!
- The *abline()* function to add the $y = x$ line. Check the function arguments on the appropriate help page.

Once you have recreated the Q-Q plot in R, use the *shapiro.test()* function to perform a test of Normality. Remember that the Shapiro-Wilk test works best for data sets with less than 50 observations, whereas the Kolmogorov-Smirnov test is generally applied for data sets with more than 50 observations. What is your conclusion with respect to the assumptions for a simple linear regression model of fish quality against delay?

9.2 Non-Parametric Tests, Fisher Exact Test and χ^2 Tests

For theory see O&L Sections 5.9, 6.3, 6.5, 8.6, 10.3 (in the 7th Edition skip McNemar Test), 10.4, 10.5 (6th and 7th Ed. [4,5]).

When making inferences about the difference between two population proportions ($\pi_1 - \pi_2$) in R the function *fisher.test()* can be used, when you need to do a Fisher Exact Test. The function calculates a p-value based on the Hypergeometric distribution. To calculate this p-value you can also use the *dhyper()* and *phyper()* functions. Remember that the Hypergeometric distribution is a discrete distribution function, so correct your outcome when necessary. For example, using Example 10.9 (6th Ed. Example 10.8) from O&L:

A clinical trial is conducted to compare two drug therapies for leukemia: P and PV. Twenty-one patients were assigned to drug P and forty-two patients to PV. Table 9.1 summarizes the successes and failures of the two drug therapies.

Table 9.1: Success and Failure outcomes for two leukemia drug therapies.

Drug	Success	Failure	Total
PV	38	4	42
P	14	7	21
Total	52	11	63

Is there significant evidence that the proportion of patients obtaining a successful outcome is higher for drug PV than for drug P?

Checking the conditions:

$$n_1\pi_1 = 38 \geq 5,\ n_1(1-\pi_1) = 4 < 5,\ n_2\pi_2 = 14 \geq 5 \text{ and } n_2(1-\pi_2) = 7 \geq 5$$

Because one of the four conditions is violated, the large sample test should not be applied. The Fisher Exact Test will be applied to this data to test the following hypotheses $H_0: \pi_P \geq \pi_{PV}$ vs. $H_a: \pi_P < \pi_{PV}$.

To calculate the p-value the hypergeometric distribution can be used in R. When x represents the number of successes for drug PV, then the expected value for the hypergeometric distribution can be calculated as $\mu = \frac{52*42}{63} = 34.667$. The p-value for the problem is then given by $P(x \geq 38) = 1 - P(x \leq 37)$. In R $P(x = 38) \approx 0.0211$ can be calculated with `dhyper(x = 38, m = 42, n = 21, k = 52)`, which is equal to `dhyper(x = 38, m = 52, n = 11, k = 42)`. $P(x \leq 37) \approx 0.9746$ can be calculated with `phyper(q = 37, m = 42, n = 21, k = 52)`, which is equal to `phyper(q = 37, m = 52, n = 11, k = 42)` (R by default uses the argument `lower.tail = TRUE`). Therefore, the p-value = $P(x \geq 38) \approx 0.0254$.

Conclusion: For all values of $\alpha \leq .025$ the p-value $\approx 0.0254 > \alpha$, there is not significant evidence that the proportion of patients obtaining a successful outcome is higher for drug PV than for drug P.

9.2.1 Exercise Non-Parametric Tests: Sign Test on Patient visit times

Recent studies of the private practices of physicians, who saw no Medicaid patients, suggested that the median length of each patient visit was 22 minutes. It is believed, that the median visit length in practices with a large Medicaid load is shorter than 22 minutes. A random sample of 20 visits in practices with a large Medicaid load yielded, in order, the following visit lengths in minutes: 9.4, 13.4, 15.6, 16.2, 16.4, 16.8, 18.1, 18.7, 18.9, 19.1, 19.3, 20.1, 20.4, 21.6, 21.9, 23.4, 23.5, 24.8, 24.9, 26.8.

Based on these data, is there sufficient evidence ($\alpha = .025$) to conclude that the median visit length in practices with a large Medicaid load is shorter than 22 minutes? To answer this question perform a Sign Test (using the *SIGN.test()* function from the **BSDA** package) and mention all steps you require to reach your conclusion.

9.2.2 Exercise Non-Parametric Tests: Sign and Wilcoxon Signed Rank Test on Albatross data

Albatrosses flying between continents: which journey is shorter, the outward journey or the homeward journey? Of eight albatrosses the number of days of their journey are measured. See data in Table 9.2 (the data is also available in `albatros_data.csv`):

Table 9.2: Outward and Homeward Journey (days) for 8 Albatrosses.

bird	outward	homeward
1	19	NA
2	28	31
3	23	25
4	20	31
5	20	33
6	24	34
7	24	21
8	22	32

a. Test for systematic difference in flying times using a sign test with $\alpha = 0.05$.

b. Calculate the test statistic for the sign test and p-value, with *pbinom()*, given by *SIGN.test()* using regular commands in the console.

c. Test for systematic difference in flying times using the Wilcoxon Signed Rank Test with $\alpha = 0.05$. For the Wilcoxon Signed Rank Test in R you can use the function *wilcox.test()* (read the help page on how to specify the function arguments). Mention all steps needed to reach your conclusion.

d. Calculate the test statistics for the Wilcoxon Signed Rank Test in R using regular commands in the console, and calculate the exact p-value using the *psignrank()* function. The *psignrank()* function provides

the Wilcoxon Signed Rank distribution function in R and gives the probability for a given quantile and sample size.

9.2.3 Exercise Non-Parametric Tests: Kruskal-Wallis test for more than 2 samples on Orange Trees

We compare the yields (in pounds) of five different varieties (A, B, C, D and E) of 4-year-old orange trees in an orchard. From each variety 7 trees are randomly sampled from this orchard.

The data for this exercise is the same as used in Section 6.2.1, were you used it in an exercise about One-Way ANOVA. The name of the data file is `orange_tree_yield.csv`.

Use the Kruskal-Wallis test (*kruskal.test()*) to test the null hypothesis H_0 that the five varieties have the same yield distributions. Use $\alpha = 0.01$. Give the null hypothesis H_0, the test statistic, outcome and p-value. Finally, give the conclusion in words.

9.2.4 Exercise χ^2 Tests: Drug Comparison

A laboratory is comparing a test drug to a standard drug against high blood pressure. In many clinical trials at many different locations, the standard drug was administered to patients with comparable hypertension. For this large patient group the fractions of patients in the four categories (1) marked decrease, (2) moderate decrease, (3) slight decrease, and (4) stationary are (0.5, 0.25, 0.1, 0.15). In a clinical trial with a random sample of 200 patients with high blood pressure, the numbers of patients in the 4 categories are given in the data set with filename `drug_comparison.csv`.

a. Read the data into R and create an object named ***pressure*** containing the column `pressure` from the data frame, that was created by reading the data into R.

b. Perform a χ^2-test with $\alpha = 0.05$ to compare the test drug with the standard drug. List all steps of the test and formulate your conclusion in words. Use the *chisq.test()* function to do a χ^2-test in R.

c. What is the correct name for this χ^2-test?

9.2.5 Exercise χ^2 Tests: Favorite Plans and Office

In a company a random sample of 216 workers is drawn. Each person selected in the sample gives his/her favorite of 3 plans for a new canteen. It is also noted in which of the 4 offices of the company the person works. This yields the following frequencies as given in Table 9.3. This data is also available in a data file named `favourite_plan.csv`.

Table 9.3: Favorite plan data for four Office locations.

Favoured plan	Office 1	Office 2	Office 3	Office 4	Total
1	15	32	18	5	70
2	8	29	23	18	78
3	1	20	25	22	68
Total	24	81	66	45	216

a. What is the full name of the χ^2-test that you have to perform in this situation?

b. Use the *table()* function in combination with the *with()* function to create a contingency table of the data.

> Tip: Adding margins in R.
>
> You can use the *addmargins()* function to display a contingency table with marginal totals. Try it! Does the end result look like Table 9.3?

c. Perform the appropriate test and give the null and alternative hypotheses as well as the test statistic, null-distribution, outcome, *p*-value and conclusion.

d. The result from the test, you performed in the previous question, contains the values for the expected counts. Can you display them in the console rounded to 1 decimal with marginal totals? This can be achieved with a single call on the console prompt.

9.3 Command Reference

Table 9.4: Lesson 9 commands

Command	Description
addmargins()	puts margins on tables or arrays
axis()	add an axis to a plot
barplot()	create a bar plot with vertical or horizontal bars
contour()	create or add a contour plot
dev.off()	shut down the specified (by default: current) device
dhyper()	hypergeometric density function
fisher.test()	Fisher's exact test for count data
image()	display three-dimensional or spatial data
jpeg()	graphics device for a jpeg format bitmap file
kruskal.test()	perform a Kruskal-Wallis rank sum test
layout()	divide a device up into rows and columns
legend()	add legends to a plot
lines()	add connected line segments to a plot
matplot()	plot columns of matrices
mtext()	write text into the margins of a plot
pdf()	start graphics device driver for a pdf vector file
persp()	draw perspective plots of a surface over the x–y plane
phyper()	hypergeometric distribution function
png()	graphics device for a png format bitmap file
points()	add points to a plot
polygon()	draw a polygon in a plot
postscript()	graphics device for a postscript vector file
psignrank()	Wilcoxon Signed Rank distribubtion function
qnorm()	calculate a quantile for a given probability in a normal distribution
qqplot()	create a Quantile-Quantile plot
quartz()	start a graphics device driver for macOS
rank()	return the sample ranks of the values in a vector
shapiro.test()	perform the Shapiro-Wilk test of Normality
SIGN.test()	sign test (**BSDA** package)
split.screen()	create/control multiple screens on a single device
text()	add text to a plot
title()	plot annotation
wilcox.test()	Wilcoxon rank sum and signed rank tests
windows()	start a graphics device driver for Windows
X11()	start a graphics device driver for X Window System

Lesson 10

End of Week 2 Assignment

This week's assignment consists of two parts (Assignment Week 2a and 2b). Write a report in R markdown, where you answer the questions given below for each part separately. Please do not put your answers and explanations in the code chunks, but use the markdown language to write your text. The code chunks are only meant for doing your calculations in R. You should write your report, in such a way, that when reading it a year from now, you yourself still understand, what you did, what the outcomes mean and what your conclusions were with respect to these outcomes.

The data are available in file `week2a_assignment.csv` and `week2b_assignment.csv`, which you can find on the course Brightspace under Course Documents → Week 2 → Data files.

Assignment Week 2a: APO-A4 gen

The response of the serum cholesterol level on the intake of cholesterol with food differs between persons. This could be due to differences in genetic sensitivity between persons. One of the genes that could play a role in this respect is the APO-A4 gene.

In an experiment, 24 persons were tested for the influence of egg consumption on the serum cholesterol level. Moreover, for each of these persons it was determined whether they were carrier of the APOA4-2 allele. The researcher classified each person according to genotype in one of three classes: 0 = non-carrier (APOA4-1/1), 1 = heterozygous carrier (APOA4-1/2), or 2 = homozygous carrier (APOA4-2/2), depending on the number of APOA4-2 alleles (0, 1 or 2). Serum cholesterol levels of these persons were measured after the diet period with additional egg consumption.

First, we want to compare the observed frequency distribution for the three genotypes with proportions 74%, 24% and 2% from the literature. To do so, we use 200 persons in the data set that were classified according to APOA4-type. This includes the aforementioned 24 persons who actually entered into the diet experiment. The 200 persons are considered to be a random sample.

a. Perform the appropriate test to test (with $\alpha = 0.05$) whether the observed frequency distribution reasonably agrees with the literature. Give the null and alternative hypotheses, outcome of the test statistic, p-value and your conclusion.

Now, we are going to analyze the data of the experiment with the 24 persons.

b. Judge by means of graphical output whether it can reasonably be assumed that the serum cholesterol level is normally distributed within each genotype group.

c. Perform an analysis of variance to test ($\alpha = 0.05$) whether a systematic difference in serum cholesterol level exists between the genotypes, assuming normality. Give the null- and alternative hypothesis, the outcome of the test statistic, distribution of the test statistic, p-value and your conclusion in words.

d. Make a table with the treatment means for the three genotypes and the associated standard errors.

e. Explain whether a comparison of mean serum cholesterol levels is required here. Regardless your answer perform the comparison.

f. Calculate Tukey's W (= difference pronounced significant according to Tukey's method) using $\alpha = 0.05$, and give the results of the pairwise comparison procedure using this W.

g. Check the model assumptions for ANOVA.

h. If the model assumptions would not have held, what would be the alternative test to test if there is difference between the non-carriers (APOA4-1/1) and the homozygous carriers (APOA4-2/2)?

Assignment Week 2b: BMI and weight in first year

The aim of some research on six year old children is to relate their BMI (**B**ody **M**ass **I**ndex, this is weight divided by the square of height in kg per m^2, and can be derived from other variables in the data set) to their weight in their first year of life. From 182 six year old children, their height (cm) and weight (kg) have been determined. From various health centers the birth weight (kg) and the weight at nine months of age (kg) of the children have been retrieved.

It is expected that BMI depends linearly upon birth weight and weight at nine months, and that the variance (σ_ε^2) in BMI does not depend on birth weight and weight at nine months. The regression model, with the usual assumptions about the error terms ε, reads as follows:

$$\text{BMI} = \beta_0 + \beta_1 \times \text{birth weight} + \beta_2 \times \text{weight at nine months} + \varepsilon$$

a. Make a matrix scatter plot of BMI, birth weight and weight at nine months. Note that BMI is not in the data file and has to be calculated first.

b. Do you think it will be difficult to get proper estimates of β_1 and β_2 because of multicollinearity? Also calculate the variance inflation factor and interpret it.

c. Fit the regression model and test if the model has any explanatory value.

d. Give the estimated regression equation.

e. Check the model assumptions by making the relevant plots and give your comments.

f. Test ($\alpha = 0.05$) whether the regression coefficient corresponding to birth weight equals 0. Give:

   1. the null and alternative hypothesis (H_0 and H_a) and what the practical meaning of the null hypothesis is,
   2. the test statistic,
   3. the distribution of the test statistic under H_0,
   4. the outcome of the test statistic,
   5. the appropriate p-value and your conclusion in words.

Part III

Week 3

Lesson 11

Getting & Cleaning Data and ANCOVA

11.1 Getting & Cleaning Data

It rarely happens that you get clean data, i.e. ready for analysis, handed to you in a project. Often you collect the data yourself in the laboratory/field or you get data from someone else, that has done experiments for you. There are many scenarios possible and data can come in many different formats.

Let's start with defining what is meant by data. A very nice definition is given on Wikipedia (https://en.wikipedia.org/wiki/Data): "Data are values of qualitative and quantitative variables, belonging to a set of items". The set of items refers to the population or a subset of it (sample), that you are interested in. Variables are the characteristics you have determined of, or measurements you have done on, units. The nature of these characteristics or measurements can be qualitative (ordinal or nominal) or quantitative (discrete or continuous).

The original data you normally start off with, is referred to as raw data. It is critical that you include the rawest form of the data that you have access to in your project folder. This ensures that data provenance (source of origin) can be maintained throughout the workflow. A few examples of raw data:

- binary file your measurement device delivers containing your measurements
- unformatted Microsoft Excel file with 10 worksheets you obtained from a colleague, containing data collected for you
- complicated JSON (JavaScript Object Notation) data you got from scraping the Twitter API (Application Programming Interface)
- hand-entered numbers collected by an experimenter looking through a microscope

You know, the raw data are in the right format, if you:

1. ran no software on the data,
2. did not modify any of the data values,
3. did not remove any data from the data set,
4. did not summarize the data in any way.

If you made any modifications of the raw data, it is not the raw form of the data. So raw data is the original form of the data, which is often hard to use for data/statistical analysis. Raw data first needs to be processed into data on which statistical analyses can be performed. The process of cleaning up the data is considered part of the analysis and should, therefore, be documented meticulously for reasons of reproducibility. All steps (merging, subsetting, transforming, etc.) should be documented. When processing standards have been used you can of course use citations to those standards.

The target for cleaning up raw data is tidy data. The general principles of tidy data have been laid out by Hadley Wickham [23]:

- each variable you measure should be in one column,

- each different observation of that variable should be in a different row,
- there should be one table for each "kind" of variable when they are originating from different sources, e.g. a table for Twitter data and a table for Facebook data,
- if you have multiple tables, there should be included a column in the table that allows them to be joined or merged (a common identifier).

While these are the hard and fast rules, there are a number of other things that will make your data set much easier to handle. First is to include a row at the top of each data table that contains variable names. Second is to make the variable names human readable, concise and meaningful, e.g. use as column name `AgeAtDiagnoses` instead of `ADx` (also see Section 6.1.1.1). Third is to save your data in one table per file.

After the cleanup process is finished you should have the following files available:

1. the raw data.
2. a tidy data set.
3. a code book describing each variable and its value in the tidy data set.
4. an explicit and exact recipe you used to go from raw data to tidy data

This lesson will give an introduction about getting data into R from various sources and cleaning it up in such a way, that you can use it for data/statistical analysis. For the cleaning process very convenient packages will be introduced. In this lesson you will also find how you should write a code book. To fulfill the last item on the list, of what you should have after cleanup, it is best to use a script, or even better a R markdown file, to describe how you got from raw to tidy data including the commands you used.

11.1.1 Getting Data

In this section you will find various ways to get data into R.

11.1.1.1 Downloading files

The internet is full of open data, that you can download and use. For downloading a file from the internet R has a function named *download.file()*, which resides in the base **utils** package. The function is useful for downloading tab-delimited a.k.a. tsv, comma-delimited a.k.a csv, and other files.

Prior to downloading a file it is good to check your current working directory and perhaps set a different working directory, using the *getwd()* and *setwd()* functions. Another approach which is often used, is to check whether there is a sub-directory `data` available. If not create it and direct the download to the created data directory. The following code checks for a `data`—directory and creates it, if not present:

```
# check for a directory "data" in your current working directory
if (!file.exists("data")) { # notice the negation symbol !
  # create a directory named "data
  dir.create(path = "data")
}
```

The following example downloads a data set, available as a `CSV`-file, about public street lighting in Wageningen:

```
fileUrl <- "https://bit.ly/2LGOOIN"
download.file(url = fileUrl,
              destfile = "./data/lights_wageningen.csv",
              method = "auto",
              mode = "wb")
dateDownloaded <- date()
writeChar(object = dateDownloaded, con = "./data/download_date_lights_wageningen.txt")
```

You could of course download the file from internet manually. However, if you write it as code in your script/R markdown file it will surely help reproducibility.

A few notes about *download.file()*:

- if the URL starts with "http://" you can most of the time use the function without specification of the `method` argument
  - if the URL starts with "https://" you may get away without specification of the `method` argument on a Windows system
  - if the URL start with "https://" you most likely need to specify `method = "curl"` or `method = "libcurl"` on a Linux/Mac system
- the function argument `mode = "wb"` ensures that the file is downloaded in binary format
- if the file, you are trying to download, is big, this function may take while
- be sure to record when you downloaded the file, as specified by the object ***dateDownloaded***.

11.1.1.2 Exercise Downloading files: Camera Data Utrecht

In this exercise you will download the same data set, about cameras placed in Utrecht on behalf of the city's Security department, in two different formats (a `CSV` and `XLS`-file). The camera data consists of a description in Dutch about the location of the camera, and the latitude and longitude of this location.

The URLs of both files are stored in the `RData`—file named `camera_resources.RData`. The location for the `CSV`-file is given in the ***fileUrlCSV*** object and the `XLS`—file in the object ***fileUrlXLS***. Write a script that:

1. checks your current working directory and saves it into an object named ***oldwd***,
2. sets the current directory to your course data directory using relative paths (see Lesson 1 about navigating directories and the use of `./` and `../`),
3. checks for the existence of a sub-directory named "`utrecht-data`" and creates it if necessary,
4. downloads both files from the internet and names them, respectively, `cameras.csv` and `cameras.xls`
5. writes the date and time, when the files have been downloaded, and prints this date and time in the console.
6. saves the date and time, when the files have been downloaded, into a text file named "`downloaded.txt`" in the created "`utrecht-data`" directory by including the following command: `writeChar(object = dateDownloaded, con = "./utrecht-data/downloaded.txt")`
7. resets your current working directory back to the position you originally started from by using the object ***oldwd***

11.1.1.3 Reading Local Flat Files

Reading local flat files into R is something, you have done quite often in the first two weeks of this course. You have used the *read.csv()* function many times. This function is a special form of the *read.table()* function in R. The *read.table()* function can be considered the main function for reading data into the random access memory of R (potentially problematic for big data files). It is a flexible and robust function, but requires specification of quite some function arguments. The most important function arguments, in order of appearance, are:

- `file`, the name of the file which the data are to be read from.
- `header`, a logical (`TRUE|FALSE`) whether the file contains the names of the variables as the first line.
- `sep`, the field separator character.
- `quote`, the set of quoting characters. The biggest trouble with reading flat files are quotation marks ' or " placed in data values, setting `quote = ""` often resolves these.
- `row.names`, a vector of row names.
- `na.strings`, a character vector that represents a missing value.
- `nrows`, an integer: the maximum number of rows to read in (*e.g.* `nrows = 10` reads 10 lines).
- `skip`, number of lines to skip before starting to read.

For a full specification of the arguments of *read.table()* check out its help page. On the help page of *read.table()* you can see that for the *read.csv()* function some default values for certain function arguments are assumed, *e.g.* `sep = ","`. The default value specification in *read.csv()* makes this function easy to use, when you know that your file complies with these default values.

11.1.1.4 Exercise Reading Local Flat Files: Camera Data Utrecht

In the previous exercise of this lesson you have downloaded the data about cameras in Utrecht into a `CSV`-file named `cameras.csv`.

a. Read this data file into R using *read.csv()* and assign it to an object named ***camerasCSV***.

b. Print the first 3 rows in the console. This is probably not what you expected. What is wrong with using *read.csv()* on your file `cameras.csv`?

c. Try reading in the data using the *read.table()* function by specifying the proper function arguments (three arguments will suffice!) and, again, print the first 3 rows in the console.

11.1.1.5 Reading Excel Files

Still, one of the most used formats for sharing data are Microsoft Excel spreadsheet files, also known as workbook files. The reason is very simple, being that a lot of people are used to using spreadsheets to collect, organize data and share those with others. Unfortunately this will probably still increase because of Google Spreadsheets, where they can be edited collaboratively over the internet. In general, it is advised to store data in either a database or in separate comma separated files (`*.csv`) or tab separated files (`*.tsv`/`*.tab`/`*.txt`) as these are easier to distribute.

To do downstream analysis on the data inside an Excel file and process the contents you first need to extract the data from the Excel files, in other words you need to read the data into R. There are several packages available to read in Excel files, among others:

1. **readxl**, this package is developed by RStudio and used when you select in the top menu of RStudio **File** → **Import Data** → **From Excel**.... Alternatively you can use the "Import Dataset" button on the *Environment* tab of RStudio and select "From Excel...".
2. **XLConnect**, the Excel Connector for R package. This package has a lot of built-in functionality for reading in and writing out Excel files in R. When you install this package, you will find an excellent vignette on the package help page, to get you started.
3. **xlsx**, provides read, write functionality for format Excel 2007 and Excel 97/2000/XP/2003 files.

Because the functionality is built-in into RStudio here an example will be shown using the **readxl** package. When you use Data Import for the first time, RStudio will probably tell you that you need to install the package first and even give you an option to do so. Once installed the data import functionality will be available also from the prompt. Assume there is an Excel spreadsheet going by the name "`experimentaldata.xlsx`", which has two worksheets: one called "experiment1" and the other called "experiment2". Then the data from "experiment2" can be read into R with the following commands:

```
library(readxl) # first load the readxl package
exp2 <- read_excel(path = "./data/experimentaldata.xlsx", sheet = "experiment2")
detach("package:readxl", unload = TRUE) # unload the readxl package
```

Here the name of the worksheet contained in the workbook was provided as named argument (`sheet`). There is also a function in the **readxl** package to extract a list of all worksheets in an Excel spreadsheet workbook file. This function is called *excel_sheets()*. The **readxl** package is currently at version 1.4.5 and contains among others the aforementioned functions.

If you want to do more, like writing objects from R into Excel spreadsheet workbooks, or reading in only certain rows and columns have a look at the **xlsx** package. For really fancy functionality with respect to Excel spreadsheets check out the **XLConnect** package, as said there is a wonderful vignette available. The following example uses **XLConnect** on fixed speed cameras and red light cameras in Baltimore. The Excel file named `trafficcameras.xlsx` was created with data from the Open Baltimore web page (https://data.baltimorecity.gov)

```
library(XLConnect)
# XLConnect 1.0.6 by Mirai Solutions GmbH [aut],
#  Martin Studer [cre],
#  The Apache Software Foundation [ctb, cph] (Apache POI),
#  Graph Builder [ctb, cph] (Curvesapi Java library),
#  Brett Woolridge [ctb, cph] (SparseBitSet Java library)
```

```
# https://mirai-solutions.ch
# https://github.com/miraisolutions/xlconnect
wb <- loadWorkbook(filename = "./data/trafficcameras.xlsx",
                   create = FALSE,
                   password = NULL)
# ERROR StatusLogger Log4j2 could not find a logging implementation. Please add
# log4j-core to the classpath. Using SimpleLogger to log to the console...
sheets <- getSheets(wb)
detach("package:XLConnect", unload = TRUE)
```

11.1.1.6 Exercise Reading Excel Files: Camera Data Utrecht

Using the **readxl** package try to read in the `cameras.xls` file, you downloaded in a previous exercise. First try to do it using the Import Data functionality of RStudio. If necessary install the **readxl** package. Next also try to do the same directly from the console by typing commands at the prompt.

11.1.1.7 Exercise Reading Excel Files: Traffic Cameras Baltimore City

For this exercise use the file `trafficcameras.xlsx` as mentioned in Section 11.1.1.5.

a. Install the **XLConnect** package with the required dependencies.

b. Execute the example given above, without unloading the package.

c. Try to extract the worksheet named "redlightcameras" into an object named ***redlights***, by finding the appropriate function in the **XLConnect** package (browse the package functions or consult the vignette).

d. Now try to unload the **XLConnect** package.

11.1.1.8 Reading SPSS Files

There are a few packages available, that allow you to read SPSS data files directly into R.

One is built by RStudio and implemented in the Import Data functionality of RStudio. This is the **haven** package, which not only reads in SPSS data files (`*.sav`) but also SAS and STATA data files as well. The package reads data into R in the `tibble` class format defined by RStudio, which means that for example factor objects will no longer be of class "`factor`" and will loose information. For example using the Drug Comparison data, used in Lesson 9 during an exercise, now provided as an SPSS `*.sav` file.

```
library(haven)
drug <- read_sav(file = "./data/drug_comparison.sav")
str(drug)
```

```
#> tibble [200 x 1] (S3: tbl_df/tbl/data.frame)
#>  $ pressure: dbl+lbl [1:200] 2, 2, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 2, 1, 1, 2, 2, ...
#>    ..@ format.spss: chr "F8.0"
#>    ..@ labels     : Named num [1:4] 1 2 3 4
#>    .. ..- attr(*, "names")= chr [1:4] "markedly" "moderately" "slightly" "stationary"
```

```
detach("package:haven", unload = TRUE)
```

Another package, which, among others, is able to read in SPSS data files (`*.sav`) into R is the **foreign** package. This package keeps the original class of objects intact, so a nominal variable in SPSS will become an object of class "`factor`" in R. Here as an example the same Drug Comparison data as an SPSS .sav file is read into R.

```
library(foreign)
drug <- suppressWarnings(read.spss(file = "./data/drug_comparison.sav"))
# Without suppressWarnings R complains about the encoding and re-encodes.
# CP-1252 is a character encoding of the Latin alphabet, used by default in the
# legacy components of Microsoft Windows in English and some other Western languages.
class(drug)
```

```
#> [1] "list"
```

```
names(drug)
```

```
#> [1] "pressure"
```

```
class(drug$pressure)
```

```
#> [1] "factor"
```

```
detach("package:foreign", unload = TRUE)
```

11.1.1.9 Exercise Reading SPSS Files: Drug Comparison

Try the examples above yourself using the file `drug_comparison.sav`.

11.1.1.10 Reading Other Files

As mentioned in Lesson 1 there are a couple thousand packages available on CRAN and Bioconductor. Among those package there are plenty to read in several types of data files. Some examples:

1. **XML** and **xml2** package, to generate and parse xml files
2. **jsonlite** and **rjson** package, to generate and parse JSON (Java Script Object Notation)
3. **sqldf** package, to perform SQL selects on R data frames
4. **RMySQL** and **RSQLite** package, to interface with MySQL from within R
5. **rhdf5** package, HDF5 (used for storing large data sets) interface to R

Search the internet for other packages you might require to read in data file types not mentioned here.

The above mentioned packages will not be discussed, so if you need to use any of them, install and look at the help pages on how to use them (always check if there is vignette included).

11.1.2 Cleaning Data

Once data have been read in or loaded into R, they need to be manipulated in order to obtain a tidy data set:

- each variable you measure should be in one column
- each different observation of that variable should be in a different row
- there should be one table for each "kind" of variable when they are originating from different sources, e.g. a table for Twitter data and a table for Facebook data,
- If you have multiple tables, they should include a column in the table that allows them to be joined or merged

In order to achieve the goal of ending up with a tidy data set, several actions might be required to achieve this. The following list sums up, far from exhaustive, some of these actions:

- subsetting,
- sorting and ordering,
- dealing with missing values,
- adding rows and columns,
- summarizing data,
- creating new variables for prediction or as predictors,
- reshaping,
- merging or joining data.

11.1.2.1 Subsetting, sorting, ordering, and dealing with missing values

By example subsetting, sorting, ordering and dealing with missing values will be covered, since most of these subjects have been discussed in previous lessons.

Let's create a data frame:

```
set.seed(13435)
x <- data.frame("var1" = sample(1:5), "var2" = sample(6:10), "var3" = sample(11:15))
x <- x[sample(1:5), ]
```

```
x$var2[c(1, 3)] <- NA
x
```

```
#>   var1 var2 var3
#> 5    2   NA   11
#> 4    4   10   12
#> 1    3   NA   14
#> 2    1    7   15
#> 3    5    6   13
```

Subsetting of x:

```
x[, 1] # subset column 1
```

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

```
# or x[, "var1"]
#  or x$var1 / x[["var1"]]
x[1:2, "var2"] # row 1 and 2 of column 2
```

```
#> [1] NA 10
```

```
# or x$var2[1:2] / x[["var2"]][1:2]
```

Subsetting using logicals, like ANDs and ORs:

```
# select row where (var1 <= 3 AND var3 > 11)
x[x$var1 <= 3 & x$var3 > 11, ]
```

```
#>   var1 var2 var3
#> 1    3   NA   14
#> 2    1    7   15
```

```
# select rows where (var1 <= 3 OR var3 > 11)
x[x$var1 <= 3 | x$var3 > 11, ]
```

```
#>   var1 var2 var3
#> 5    2   NA   11
#> 4    4   10   12
#> 1    3   NA   14
#> 2    1    7   15
#> 3    5    6   13
```

Dealing with missing values:

```
# Delete rows where var2 has NA values
x[!is.na(x$var2), ]
```

```
#>   var1 var2 var3
#> 4    4   10   12
#> 2    1    7   15
#> 3    5    6   13
```

```
# Use which() function
range(x$var2, na.rm = TRUE)
```

```
#> [1]  6 10
```

```
x[which(x$var2 >= range(x$var2, na.rm = TRUE)[1]), ]
```

```
#>   var1 var2 var3
#> 4    4   10   12
#> 2    1    7   15
#> 3    5    6   13
```

Remind yourself that subsetting on `NA` values, does not produce the actual rows. So constructs like `x[which(x\$var2 == NA), ]` wil **not** work.

Sorting and ordering has not been discussed so far. Sorting with the *sort()* function can be applied on a vector or factor object. When assigning a sorted column/row back to a matrix or data frame only that specific column/row will be sorted. For example:

```
sort(x$var1) # ascending by default
```

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

```
sort(x$var1, decreasing = TRUE) # sort descending
```

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

```
x2 <- x
x2$var1 <- sort(x2$var1)
x2
```

```
#>   var1 var2 var3
#> 5    1   NA   11
#> 4    2   10   12
#> 1    3   NA   14
#> 2    4    7   15
#> 3    5    6   13
```

When you want to sort the whole matrix or data frame, keeping the row information intact, you have to use the *order()* function. The *order()* returns a vector indicating the position of the sorted values. This vector can be used as an integer index vector for subsetting. For example:

```
x
```

```
#>   var1 var2 var3
#> 5    2   NA   11
#> 4    4   10   12
#> 1    3   NA   14
#> 2    1    7   15
#> 3    5    6   13
```

```
order(x$var3) # sorted positions of x$var3
```

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

```
x[order(x$var3), ] # x sorted on column x$var3
```

```
#>   var1 var2 var3
#> 5    2   NA   11
#> 4    4   10   12
#> 3    5    6   13
#> 1    3   NA   14
#> 2    1    7   15
```

The *order()* function can be used to sort rows for multiple columns sequentially, *e.g.*

```
y <- data.frame(level1 = c(rep(1, 3), rep(2, 3)),
                level2 = letters[1:6],
                level3 = LETTERS[26:21])
set.seed(1492)
(y <- y[sample(1:6), ])
```

```
#>   level1 level2 level3
#> 4      2      d      W
#> 2      1      b      Y
#> 6      2      f      U
#> 1      1      a      Z
#> 3      1      c      X
```

```
#> 5      2      e      V
```

```
y[order(y$level1), ] #sort rows by level1
```

```
#>   level1 level2 level3
#> 2      1      b      Y
#> 1      1      a      Z
#> 3      1      c      X
#> 4      2      d      W
#> 6      2      f      U
#> 5      2      e      V
```

```
y[order(y$level1, y$level2), ] # sort rows by level1, next level2 to get original y
```

```
#>   level1 level2 level3
#> 1      1      a      Z
#> 2      1      b      Y
#> 3      1      c      X
#> 4      2      d      W
#> 5      2      e      V
#> 6      2      f      U
```

11.1.2.2 Adding Rows and Columns

Rows can be added directly to matrices and data frames, but be aware that by accident no existing row or rows are overwritten with new values. Also, make sure that the added values matches the dimension of the object you are adding to , *e.g.*

```
print(x)
```

```
#>   var1 var2 var3
#> 5    2   NA   11
#> 4    4   10   12
#> 1    3   NA   14
#> 2    1    7   15
#> 3    5    6   13
```

```
dim(x)
```

```
#> [1] 5 3
```

```
(x[nrow(x) + 1, ] <- c(1, 2, 3)) # add a row to x
```

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

```
dim(x)
```

```
#> [1] 6 3
```

```
x$var4 <- rnorm(dim(x)[1])
x # autoprint x
```

```
#>   var1 var2 var3        var4
#> 5    2   NA   11 -0.27828867
#> 4    4   10   12  0.36249296
#> 1    3   NA   14 -0.85986275
#> 2    1    7   15  0.05421515
#> 3    5    6   13 -1.15820705
#> 6    1    2    3  0.41475084
```

A safer way of adding rows and columns is using *rbind()* and *cbind()*, for these functions add rows and columns at the far end of the matrix or data frame. Still, you will have to keep the dimensions of the object you are binding to in mind, in order to provide enough to fill the row or column.

11.1.2.3 Summarizing Data

The key process of data cleaning is looking at the data set that has been loaded into R, and identifying any sort of quirks or weird issues or missing values, or other problems that you need to address before you can do downstream analysis. So the first thing that you want to be able to do is summarize the data that have just been loaded in.

In this section about summarizing data the data set used contains data about restaurants in Baltimore. The reason in this course so many data sets are coming from Baltimore, is that the government in that city has an excellent open data base with all sorts of data that can be publicly used (https://data.baltimorecity.gov). First the data need to be read into R,

```
restdata <- read.csv(file = "./data/restaurants.csv")
```

First eyeball the data to get a feeling, instead of printing everything in the console, look only at the first and last three rows.

```
head(restdata, n = 3)
```

```
#>    name zipCode neighborhood councilDistrict policeDistrict
#> 1   410   21206    Frankford              2   NORTHEASTERN
#> 2  1919   21231  Fells Point              1   SOUTHEASTERN
#> 3 SAUTE   21224       Canton              1   SOUTHEASTERN
#>                          Location.1
#> 1 4509 BELAIR ROAD\nBaltimore, MD\n
#> 2    1919 FLEET ST\nBaltimore, MD\n
#> 3   2844 HUDSON ST\nBaltimore, MD\n
```

```
tail(restdata, n = 3)
```

```
#>                  name zipCode  neighborhood councilDistrict policeDistrict
#> 1325 ZINK'S CAF\u0090   21213 Belair-Edison              13   NORTHEASTERN
#> 1326     ZISSIMOS BAR   21211       Hampden               7       NORTHERN
#> 1327           ZORBAS   21224     Greektown               2   SOUTHEASTERN
#>                             Location.1
#> 1325 3300 LAWNVIEW AVE\nBaltimore, MD\n
#> 1326      1023 36TH ST\nBaltimore, MD\n
#> 1327  4710 EASTERN Ave\nBaltimore, MD\n
```

With the *summary()* function an overall summary of the data frame in use can be obtained:

```
summary(restdata)
```

```
#>      name              zipCode       neighborhood       councilDistrict
#>  Length:1327        Min.   :-21226   Length:1327        Min.   : 1.000
#>  Class :character   1st Qu.: 21202   Class :character   1st Qu.: 2.000
#>  Mode  :character   Median : 21218   Mode  :character   Median : 9.000
#>                     Mean   : 21185                      Mean   : 7.191
#>                     3rd Qu.: 21226                      3rd Qu.:11.000
#>                     Max.   : 21287                      Max.   :14.000
#>  policeDistrict      Location.1
#>  Length:1327        Length:1327
#>  Class :character   Class :character
#>  Mode  :character   Mode  :character
#>
#>
#>
```

The summary gives for each column in the data frame object information about its content. For columns of class `character` the number of rows is displayed as well as the class and mode. For numerical columns the Tukey five number summary is given, e.g. there are 14 districts in Baltimore (maximum of `councilDistrict`). You can also see that the zip code is treated as a numerical value, and that there is at least one zip code wrongly assigned with a negative value.

Instead of making a summary the *str()* can also be called. This will provide direct information about the number of rows and columns in the data frame and the class of each column.

To find the wrongly assigned zip codes a table of the zip codes can be useful:

```
table(restdata$zipCode, useNA = "ifany")
```

```
#>
#> -21226  21201  21202  21205  21206  21207  21208  21209  21210  21211  21212
#>      1    136    201     27     30      4      1      8     23     41     28
#>  21213  21214  21215  21216  21217  21218  21220  21222  21223  21224  21225
#>     31     17     54     10     32     69      1      7     56    199     19
#>  21226  21227  21229  21230  21231  21234  21237  21239  21251  21287
#>     18      4     13    156    127      7      1      3      2      1
```

See that there is only one wrongfully assigned zip code with a negative value. The table also shows zip code 21202 has the most restaurants in the Baltimore city area.

Another nice way to summarize the data is by creating two-way tables, for example showing how many restaurants are per zip code in what district:

```
head(table(restdata$zipCode, restdata$councilDistrict)) # showing only the first 6 rows
```

```
#>
#>           1   2   3   4   5   6   7   8   9  10  11  12  13  14
#>   -21226  0   0   0   0   0   0   0   0   0   1   0   0   0   0
#>   21201   0   0   0   0   0   0   0   0   1   0 115  20   0   0
#>   21202  37   0   0   0   0   0   0   0   0   1 139  24   0   0
#>   21205   0   3   0   0   0   0   0   0   0   0   0   4  20   0
#>   21206   0  27   0   0   0   0   0   0   0   0   0   0   3   0
#>   21207   0   0   0   0   3   0   0   1   0   0   0   0   0   0
```

Or create tables with specific characteristics

```
table(restdata$councilDistrict %in% c(9))
```

```
#>
#> FALSE  TRUE
#>  1252    75
```

```
table(restdata$zipCode %in% c(21212, 21213))
```

```
#>
#> FALSE  TRUE
#>  1268    59
```

These specific characteristics can also be used again, to obtain the full information about the restaurant fulfilling the specific characteristic by means of subsetting.

Contingency tables can also be a useful way of summarizing data. In the following examples the built-in data sets ***UCBAdmissions*** and ***warpbreaks*** will be used to illustrate the use of crosstabs (contingency tables).

```
dfUCBAdmissions <- as.data.frame(datasets::UCBAdmissions)
warpbreaks <- datasets::warpbreaks
```

Looking at the structure of both objects, reveals that the data frame ***dfUCBAdmissions*** actually has three factors: `Admit` with 2 levels, `Gender` with 2 levels, `Dept` with 6 levels, and that ***warpbreaks*** has two factors: `wool` with 2 levels and `tension` with 3 levels. However, ***warpbreaks*** has 54 observations, in order to see what is going on make a table of tension versus wool for the warpbreaks data:

```
table(warpbreaks$tension, warpbreaks$wool)
```

```
#>
#>     A B
#>   L 9 9
#>   M 9 9
```

```
#>   H 9 9
```

The table shows that tension is replicated 9 times for each wooltype.

Now let us generate some contingency tables for these two data sets, for the ***dfUCBAdmissions*** data frame a contingency table is created showing the admitted and rejected number of male and female student over all departments, and for the warpbreaks data set contingency tables are created for the breaks per wool type at each tension level per replicate.

```
(xtAdmissions <- xtabs(Freq ~ Gender + Admit, data = dfUCBAdmissions))
```

```
#>         Admit
#> Gender   Admitted Rejected
#>   Male       1198     1493
#>   Female      557     1278
```

```
warpbreaks$replicate <- rep(1:9, length.out = 54)
xtbreaks <- xtabs(breaks ~., data = warpbreaks)
```

The formula for generating the crosstabs for the warpbreaks data set specifies that crosstabs of the breaks should be made for all other columns. The ***xtbreaks*** object was not printed, because the output would be an array of 9 contingency tables (one for each replicate). Instead of printing the array of crosstabs, the array can also be flattened and printed,

```
ftable(xtbreaks)
```

```
#>              replicate  1  2  3  4  5  6  7  8  9
#> wool tension
#> A    L                 26 30 54 25 70 52 51 26 67
#>      M                 18 21 29 17 12 18 35 30 36
#>      H                 36 21 24 18 10 43 28 15 26
#> B    L                 27 14 29 19 29 31 41 20 44
#>      M                 42 26 19 16 39 28 21 39 29
#>      H                 20 21 24 17 13 15 15 16 28
```

11.1.2.4 Creating New Variables

In the previous section in the ***warpbreaks*** example a new factor variable was created to deal with replication in the data set. Often raw data will not have a value you are looking for, or you need to transform the data to get the values you need. Usually you will add those values for the new variables to the data frame you are working with, e.g. variables to be used for prediction or variables to be used as predictors.

Some common variables to create are:

- missing value indicators
- "cutting up" quantitative variables (subsetting)
- applying transformations

Common techniques for creating new variables are:

- generating sequences
- subsetting
- logical indexing (`TRUE`|`FALSE`)
- initiating factor variables

11.1.2.5 Reshaping Data

Sometimes it is necessary to reshape your data to get a usable data set. One example you have seen, just before, when the array of contingency tables was flattened. Reshaping your data also becomes very important, when creating, so-called, Trellis plots. These are windowed plots, where each window plot is made for a certain condition, e.g. plot y versus x given condition z. Both the **lattice** and **ggplot2** packages typically require this type of reshaped data sets in order to make conditional plots.

A very good package to reshape your data is the **reshape2** package, and after using it for a while you will truly appreciate its functionality.

Here some examples will be given using the ***mtcars*** data set, you have seen before in previous lessons.

```
head(mtcars, n = 3)
```

```
#>                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
```

```
mtcars[["carname"]] <- rownames(mtcars) # assign rownames to a column "carname"
library(reshape2)
carMelt <- melt(data = mtcars,
                id.vars = c("carname", "gear", "cyl"),
                measure.vars = c("mpg", "hp"))
head(carMelt, n = 3)
```

```
#>         carname gear cyl variable value
#> 1     Mazda RX4    4   6      mpg  21.0
#> 2 Mazda RX4 Wag    4   6      mpg  21.0
#> 3    Datsun 710    4   4      mpg  22.8
```

```
tail(carMelt, n = 3)
```

```
#>          carname gear cyl variable value
#> 62  Ferrari Dino    5   6       hp   175
#> 63 Maserati Bora    5   8       hp   335
#> 64    Volvo 142E    4   4       hp   109
```

The effect of the reshape *melt()* function is that it turned the wide table into a melted, so-called, tall table.

After melting a data frame it can be reshaped into another form by casting it into a new mold (sounds a bit like melting and casting iron, well the terminology was taken from it!). For example,

```
(cylData <- dcast(data = carMelt, formula = cyl ~ variable))
```

```
#>   cyl mpg hp
#> 1   4  11 11
#> 2   6   7  7
#> 3   8  14 14
```

The casted data frame ***cylData*** displays the number of cars with 4, 6 and 8 cylinders. Another option, that can be very useful, is to use an aggregate function,

```
(cylDataMean <- dcast(data = carMelt,
                      formula = cyl ~ variable,
                      fun.aggregate = mean))
```

```
#>   cyl      mpg        hp
#> 1   4 26.66364  82.63636
#> 2   6 19.74286 122.28571
#> 3   8 15.10000 209.21429
```

Instead of reshaping and casting data frames, the split-apply-combine approach could probably also be used to get the same result.

11.1.2.6 Merging or Joining Data

To merge two data frames (data sets) horizontally, use the *merge()* function. The most important function arguments are:

- `x`, `y` supplying the two data frames to be merged
- `by`, `by.x`, `by.y` specifications of the columns used for merging.
- `all`, a logical; `all = L` is shorthand for `all.x = L` and `all.y = L`, where `L` is either `TRUE` or `FALSE`.

By default, the *merge()* function merges all common column names. In most cases, you join two data frames by one or more common key variables (*i.e.*, an inner join).

An example of merging data sets:

```
(x <- data.frame(k1 = c(NA, NA, 3, 4, 5),
                 k2 = c(1, NA, NA, 4, 5),
                 data = 1:5))
```

```
#>   k1 k2 data
#> 1 NA  1    1
#> 2 NA NA    2
#> 3  3 NA    3
#> 4  4  4    4
#> 5  5  5    5
```

```
(y <- data.frame(k1 = c(NA, 2, NA, 4, 5),
                 k2 = c(NA, NA, 3, 4, 5),
                 data = 1:5))
```

```
#>   k1 k2 data
#> 1 NA NA    1
#> 2  2 NA    2
#> 3 NA  3    3
#> 4  4  4    4
#> 5  5  5    5
```

```
merge(x, y, by = c("k1", "k2")) # NA's match
```

```
#>   k1 k2 data.x data.y
#> 1  4  4      4      4
#> 2  5  5      5      5
#> 3 NA NA      2      1
```

```
merge(x, y, by = "k1") # NA's match, so 6 rows
```

```
#>   k1 k2.x data.x k2.y data.y
#> 1  4    4      4    4      4
#> 2  5    5      5    5      5
#> 3 NA    1      1   NA      1
#> 4 NA    1      1    3      3
#> 5 NA   NA      2   NA      1
#> 6 NA   NA      2    3      3
```

```
merge(x, y, by = "k2", incomparables = NA) # 2 rows
```

```
#>   k2 k1.x data.x k1.y data.y
#> 1  4    4      4    4      4
#> 2  5    5      5    5      5
```

To join two data frames (data sets) vertically, use the *rbind()* function.

11.1.2.7 Exercise SWIRL: Getting & Cleaning Data

In the introduction was stated that you would get to learn some great packages for Getting & Cleaning Data. You have been provided with and have worked with a bunch of nice packages regarding Getting Data. So now it is time to keep up with the promise with respect to Cleaning Data. Instead of just telling about them and giving examples, you get to work with them right away.

In this exercise you will do 3 lessons in SWIRL about Getting & Cleaning Data. However, those lessons are no longer contained in the "R Programming" SWIRL course you installed at the beginning of "R for Statistics". You will have to install another SWIRL course.

Download the file `Getting_and_Cleaning_Data.zip` from Brightspace (available as file at the top in Week 3,

as well as in Lesson 11 under "Installation SWIRL Course: Getting and Cleaning Data"). Next perform the following steps:

1. Open RStudio
2. Execute in the console the R command: `.libPaths()` to show the path, where packages are installed (it will be the one shown after `[1]` in the console)
3. Open File Explorer/Finder and navigate to the path, where packages are installed as determined in step 2.
4. Open the sub-folder `Courses` inside the `swirl` folder.
5. Copy the contents of the downloaded zip-file into the `Courses` sub-folder.

Now load the **swirl** package in RStudio using: `library(swirl)`. After executing in the console: `swirl()`, start the lessons from the **Getting and Cleaning Data** course. First do the two lessons about the **dplyr** package. In those two lectures you will learn how to use this great package for cleaning data.

Once you finished with those two lessons, continue with the lesson about "Tidying Data with tidyr". In this lesson you will learn to use the **tidyr** package for tidying data into a tidy data set. You will need what you have learned about the **dplyr** package as well in this swirl lesson.

11.1.3 The Code Book

For almost any data set, the measurements you calculate will need to be described in more detail than you can or should sneak into a spreadsheet. The code book contains this information. At minimum it should contain:

1. Information about the variables (including units!) in the data set not contained in the tidy data,
2. Information about the summary choices you made,
3. Information about the experimental study design you used.

Here is an example of how this would work from genomics. Suppose that for 20 people you have collected gene expression measurements with RNA-sequencing (https://en.wikipedia.org/wiki/RNA-Seq). You have also collected demographic and clinical information about the patients including their age, treatment, and diagnosis. You would have one table/spreadsheet that contains the clinical/demographic information. It would have four columns (patient id, age, treatment, diagnosis) and 21 rows (a row with variable names, then one row for every patient). You would also have one spreadsheet for the summarized genomic data. Usually this type of data is summarized at the level of the number of counts per exon. Suppose you have 100,000 exons, then you would have a table/spreadsheet that had 21 rows (a row for gene names, and one row for each patient) and 100,001 columns (one column for the patient id and one column for each exon).

In this genomics example, the analyst would want to know what the unit of measurement for each clinical/demographic variable is (age in years, treatment by name/dose, level of diagnosis and how heterogeneous). They would also want to know how you picked the exons you used for summarizing the genomic data (UCSC/Ensembl, *etc.*). They would also want to know any other information about how you did the data collection/study design. For example, are these the first 20 patients that walked into the clinic? Are they 20 highly selected patients by some characteristic like age? Are they randomized to treatments?

A common format for this document is a Word file. There should be a section called "Study design" that has a thorough description of how you collected the data. There is a section called "Code book" that describes each variable and its units.

11.1.3.1 How to code variables

When you put variables into a spreadsheet there are several main categories you will run into depending on their data type (https://en.wikipedia.org/wiki/Statistical_data_type:

1. Continuous
2. Ordinal
3. Categorical
4. Missing
5. Censored

Continuous variables are anything measured on a quantitative scale that could be any fractional number. An example would be something like weight measured in kg. Ordinal data (https://en.wikipedia.org/wiki/Ordinal_data) are data that have a fixed, small (< 100) number of levels but are ordered. This

could be for example survey responses where the choices are: poor, fair, good. Categorical data (https://en.wikipedia.org/wiki/Categorical_variable) are data where there are multiple categories, but they aren't ordered. One example would be sex: male or female. This coding is attractive because it is self-documenting. Missing data (https://en.wikipedia.org/wiki/Missing_data) are data that are unobserved and you don't know the mechanism. You should code missing values as `NA`. Censored data (https://en.wikipedia.org/wiki/Censoring_(statistics)) are data where you know the missingness mechanism on some level. Common examples are a measurement being below a detection limit or a patient being lost to follow-up. They should also be coded as `NA` when you don't have the data. But you should also add a new column to your tidy data called, "VariableNameCensored" which should have values of "`TRUE`" if censored and "`FALSE`" if not. In the code book you should explain why those values are missing. It is absolutely critical to report to the analyst if there is a reason you know about that some data are missing. You should also not impute (https://en.wikipedia.org/wiki/Imputation_(statistics))/ make up / throw away missing observations.

In general, try to avoid coding categorical or ordinal variables as numbers. When you enter the value for sex|gender in the tidy data, it should be "male" or "female". The ordinal values in the data set should be "poor", "fair", and on the other extreme "good" not 1, 2 ,3. This will avoid potential mix-ups about which direction effects go and will help identify coding errors.

Always encode every piece of information about your observations using text. For example, if you are storing data in Excel and use a form of colored text or cell background formatting to indicate information about an observation ("red variable entries were observed in experiment 1.") then this information will not be exported (and will be lost!) when the data is exported as raw text. Every piece of data should be encoded as actual text that can be exported.

11.1.4 The instruction list/script

You may have heard this before, but reproducibility is a big deal in computational science (https://www.science.org/doi/10.1126/science.1213847). That means, when you submit your paper, the reviewers and the rest of the world should be able to exactly replicate the analyses from raw data all the way to final results. If you are trying to be efficient, you will likely perform some summarization/data analysis steps before the data can be considered tidy.

The ideal thing for you to do when performing summarization is to create a computer script (in `R`, `Python`, or something else) that takes the raw data as input and produces the tidy data you are sharing as output. You can try running your script a couple of times and see if the code produces the same output.

In many cases, the person who collected the data has incentive to make it tidy for a statistician to speed the process of collaboration. They may not know how to code in a scripting language. In that case, what you should provide the statistician is something called pseudo-code (https://en.wikipedia.org/wiki/Pseudocode). It should look something like:

Step 1 Take the raw file, run version 3.1.2 of summarize software with parameters a = 1, b = 2, c = 3
Step 2 Run the software separately for each sample
Step 3 Take column three of outputfile.out for each sample and that is the corresponding row in the output data set

You should also include information about which system (Mac/Windows/Linux) you used the software on and whether you tried it more than once to confirm it gave the same results. Ideally, you will run this by a fellow student/lab-mate/colleague to confirm that they can obtain the same output file you did.

11.2 ANCOVA

For theory see O&L Sections 12.1, 12.2, 12.7, 16.1, 16.2, 16.3 (6^{th} and 7^{th} Ed. [4,5]).

ANCOVA is an acronym for Analysis of Covariance. You may look at ANCOVA in two ways: ANCOVA in broad sense or in narrow sense.

- An ANCOVA model in broad sense is a linear model in which there is at least one qualitative regressor and one quantitative regressor. Typically, qualitative regressors (factors) are coded by dummy 0/1 variables, as R does in default parameteriziation. You end up with linear models with multiple regression

lines, which may have different slopes and intercepts, may have a common slope (parallel lines), or may have a common the colleagueintercept.

- ANCOVA in the narrow sense describes the analysis of an experiment in which the main interest is in a treatment factor, but we wish to correct for a quantitative covariate, like the starting weight of an animal. The model is a parallel lines regression model. The covariate is measured to obtain higher precision for the comparison of treatments.

Whether you look at ANCOVA in the broad sense or narrow sense, all the needed R functions for analysis were already studied earlier.

In two exercises we will look at examples of ANCOVA in the broad sense and ANCOVA in the narrow sense.

11.2.1 Exercise ANCOVA in broad sense: pH and lime

In a study on the effect of lime upon the pH of soils, different amounts of lime were applied to samples from a soil. The amounts were 0, 1, 2, 4, and 8 units (not further specified). There were two types of lime: Agricultural lime (A) and Granular lime (B).

- Read the data file `lime.csv` into R, and show the first few lines.
- Make sure that the variable named f is a factor.
- Give the following names to the levels of this factor: "AL","GL" using the function *levels()*.
- Make a scatterplot of soil pH versus lime amount, while making a distinction between the two types of lime (either by coloration or different plotting symbols).

We are interested in the relationship between amount of lime and pH, separately for the two types of lime.

- Fit simple linear regression models to the agricultural lime data and granular lime data separately, and write down the fitted regression equations for the two groups.

With this approach it is not possible to statistically determine whether the slopes for the two regression lines are different. We need to fit a single multiple regression model, that will allow two regression lines with different intercepts and slopes for the two groups.

- Fit a single multiple linear regression models to the combined data. Use model formula `y ~ f + x + f:x` within the *lm()* function. Give a summary of the fitted regression model, paying special attention to the estimated regression coefficients. What do they represent?
- Write down the estimated regression equations, separately for agricultural and granular lime, based upon the fitted multiple regression model. You should combine some coefficients to get the intercept and slope in the granular lime group. Do you find the same regression equations as found with two separate simple linear regressions?

We can obtain exactly the same output by coding a dummy variable and product variable ourselves and use these in the regression model.

- Calculate a dummy variable named ***dGL*** for the second level of the factor. This dummy should have value 1 for an observation from the GL group, and 0 otherwise.
- Calculate the product between this dummy and variable x, and name it ***xGL***.
- Next, fit the multiple regression model using ***dGL***, ***x*** and ***xGL*** as regressors and compare with the earlier output.
- What is the regression coefficient of ***xGL*** telling?
- Test the null hypothesis that this coefficient is zero with a t-test.
- Use the *anova()* function to obtain an F-test for the same coefficient. Are the P-values of the F-test and t-test identical?
- Test the null hypothesis that the intercepts are the same with a t-test.
- Now test the null hypothesis with an F-test. For this, fit a Reduced Model, without the dummy ***dGL*** (but with the product ***xGL***) and compare Full and Reduced models with the *anova()* function.

- Why does it make sense that the two intercepts are identical?

If all is well, you concluded that different slopes are necessary, but a common intercept makes more sense than different intercepts.

- Fit the final regression model with different slopes, but a common intercept.
- Write down the estimated regression equations for agricultural and granular lime, based upon the fitted regression model.

11.2.2 Exercise ANCOVA in narrow sense: Effect of Fertilizer on Peanut Seed Yield

We look at example 16.1 from O&L 6^{th} or 7^{th} Edition about the seed yield (in grams) of peanut plants. Three treatments were compared: a control fertilizer (C), a slow-release fertilizer (S) or fast-release fertilizer (F). Ten replications of each treatment were grown in a greenhouse. The 30 peanut plants were not exactly at the same level of development at the start of the study. Therefore, the researcher recorded the height (in cm) of the plants.

- Read the data file `peanuts.csv` into R, and show the first few lines.
- Make sure that the treatment variable named `f` is a factor.
- Use the *rename()* function from the **dplyr** package to rename the variables, giving them more informative names.
- Make a scatterplot of seed yield versus height, using different colors or plotting symbols for the three treatments).

The traditional ANCOVA model (parallel lines) is as follows:

$$y_{ij} = \mu + \alpha_i + \beta x_{ij} + \epsilon_{ij}$$

with i the index for fertilizer type ($i = 1, 2, 3$), j is index for replicate plant within treatment group ($j = 1, \ldots, 10$), x is the covariate plant height. Realize that this model specifies three parallel regression lines.

- Fit a traditional ANCOVA model (parallel lines), explaining seed yield by fertilizer and correcting for plant height.
- Study the output from the model by the use of the *summary()* and *anova()* functions.
- Is the regression coefficient for the covariate height significantly different from zero?

We can judge the performance of the ANCOVA model, by comparing with the one-way ANOVA model using the relative efficiency. The relative efficiency is obtained by comparing the mean squared errors (MSE) for the two models. If the ANCOVA is better than the one-way ANOVA model, the ANCOVA's MSE is (substantially) smaller than the one-way ANOVA's MSE. The relative efficiency is (roughly) the ratio of the two, and tells how much larger the sample size needs to be to get the same precision in the one-way ANOVA model, as obtained now with the ANCOVA model.

- Now fit the one-way ANOVA model, obtain the MSE's (Mean Square Errors) from the ANCOVA and from the one-way ANOVA models, and calculate the relative efficiency. What do you conclude?

Comparing the treatments, now corrected for the covariate, can be done as before.

- load the **emmeans** package.
- Calculate and show the corrected means of seed yield, corrected for height using the R code (replacing the names of the R objects ***lmo*** and ***f*** with the names you used): `lsmPeanut <- emmeans(lmo, "f"))`.
- Check that these corrected means can be obtained by using the coefficients from the fitted ANCOVA model: $\hat{\mu} + \hat{\alpha}_i + \hat{\beta}\bar{x}$. To calculate these corrected means, the estimated regression coefficients are needed, and the overall average of plant height.
- Pairwise comparison among the means are obtained as: `pairs(lsm.peanut, adjust = "none")`.

Lesson 12

Simulation, Experimental Design, Power Analysis and Sample Sizes

12.1 Simulation

Numerical simulation is an important topic, since it can be used for:

- Figuring out what to expect.
- Testing a hypothesis.
- Experimenting with a model structure.
- Analyzing complex systems.

Within a simulation you usually generate data using random numbers with known characteristics and which follow hypothesized processes. You can collect vast amounts of virtual, *i.e.* computer generated, data to test out hypotheses. With R this is quite easy, as it contains functions for simulating numbers or variables from given probability distributions.

12.1.1 Probability distributions

In this course so far you have seen already some functions for probability distributions in R, *e.g.*

- *rnorm()* to generate random Normal variates with a given mean and standard deviation.
- *pbinom()* to evaluate the cumulative binomial distribution function $P(y \leq k)$.
- *qt()* to look up quantiles from a t-distribution with a given probability and degrees of freedom.

There are generally within R four functions associated with any distribution, which you can recognize by their prefixes:

- **d** for density
- **r** for random number generation
- **p** for cumulative probability density
- **q** for quantile function

R contains many distributions, some of those are shown in Table 12.1 with their additional parameters.

Table 12.1: Some distributions in R with their arguments.

Distribution	R function	Additional arguments
Beta	**beta()*	`shape1, shape2, ncp`
Binomial	**binom()*	`size, prob`
Cauchy	**cauchy()*	`location, scale`

Distribution	R function	Additional arguments
Chi-squared	**chisq()*	`df, ncp`
Exponential	**exp()*	`rate`
F	**f()*	`df1, df2, ncp`
Gamma	**gamma()*	`shape, scale`
Geometric	**geom()*	`prob`
Hypergeometric	**hyper()*	`m, n, k`
Log-normal	**lnorm()*	`meanlog, sdlog`
Logistic	**logis()*	`location, scale`
Negative binomial	**nbinom()*	`size, prob`
Normal	**norm()*	`mean, sd`
Poisson	**pois()*	`lambda`
Signed rank	**signrank()*	`n`
Student t	**t()*	`df, ncp`
Studentized range	**tukey()*	`nmeans, df`
Uniform	**unif()*	`min, max`
Weibull	**weibull()*	`shape, scale`
Wilcoxon Rank Sum Statistic	**wilcox()*	`m, n`
Wilcoxon Signed Rank Statistic	**signrank()*	`n`

As you can see in Table 12.1, different distributions come with different parameters. This even depends on the prefix you are using! Please check the documentation for all possible parameters of each probability distribution.

12.1.2 Setting the seed

When generating random numbers from any distribution its very important to control the starting put of the random number generation. If you need to rerun your calculations you would like your random numbers to be drawn in the same way, as the first time you ran your simulation.

Within R this is controlled by setting the seed with the *set.seed()*. This function controls the starting point of random number generation. Because random number generation in a computer is controlled by a starting point, you will understand why this actually called pseudo random number generation. Although the generated numbers are not truly random, it helps tremendously when it comes to reproducibility of your simulations.

12.1.3 Random Sampling

Random sampling is another aspect that is commonly used in simulations. R has the function *sample()*, which you can apply for drawing random samples and generate permutations.

Recall that R has a built-in constant `letters`, which contains the 26 lower-case letters of the Roman alphabet. You can ask R to give a random letter using:

```
set.seed(seed = 1492) # to ensure the same result
sample(x = letters, size = 1)
```

```
#> [1] "t"
```

You can also use the *sample()* function with replacement (the default is without replacement). For example, you could generate a vector of 100 random letters using:

```
set.seed(seed = 1492) # to ensure the same result
sample(x = letters, size = 100, replace = TRUE)
```

```
#>   [1] "t" "p" "v" "j" "i" "g" "q" "b" "z" "h" "h" "i" "g" "w" "m" "b" "e" "t"
#>  [19] "t" "b" "y" "u" "t" "p" "m" "f" "f" "f" "y" "d" "m" "o" "h" "o" "o" "t"
#>  [37] "n" "f" "x" "r" "y" "d" "v" "h" "c" "f" "f" "k" "l" "x" "p" "f" "w" "l"
#>  [55] "s" "v" "b" "v" "t" "s" "m" "y" "f" "a" "t" "a" "e" "v" "s" "x" "i" "x"
#>  [73] "c" "m" "p" "o" "l" "c" "m" "j" "p" "v" "l" "i" "h" "s" "a" "p" "o" "f"
#>  [91] "f" "f" "c" "h" "i" "e" "w" "t" "e" "u"
```

A nice feature of the *sample()* function is, that it can also be used to permute data. For example:

```
(sampleList <- seq(from = 1, to = 10, by = 1))
```

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

```
set.seed(seed = 1492)
(permutedList <- sample(x = sampleList))
```

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

12.1.4 Exercise SWIRL: Simulation

Start *swirl()* from the **swirl** package, as you have done before, and work your way through lesson 13 on Simulation.

12.1.5 Exercise Simulation: A linear model

In the slides you have seen an example how to simulate from a linear model. In this exercise you will perform your own simulation from this framework:

$$y = \beta_0 + \beta_1 \times x + \varepsilon$$

with $\varepsilon \sim N(0, \sigma^2)$. Choose the regression coefficients β_0 and β_1 as well as σ. Simulate x, being uniformly distributed on $[0, 1]$, with length n.

Run the simulation for sample sizes $n = 10,\ 50,$ and100. Check with the *lm()* function, how well you can recover the model underlying the simulation and make diagnostic plots to check the assumptions of a linear model.

12.1.6 Exercise Simulation: Coin toss

Write a script to simulate tossing a (fair) coin and record the proportion of heads (so far). Simulate $n = 100$ tosses and plot the proportion of heads, that you observed.

12.1.7 Properties of Means and Variances

When x, y are random variables and $a, b, c \in \mathbb{R}$, then the population mean (a.k.a. the expected value) of $ax + by + c$ can be calculated with:

$$\mathrm{E}(a \times x + b \times y + c) = a \times \mathrm{E}(x) + b \times \mathrm{E}(y) + c$$

or equivalently

$$\mu_{a \times x + b \times y + c} = a \times \mu_x + b \times \mu_y + c$$

When x, y are **independent** random variables and $a, b, c \in \mathbb{R}$, then the population variance of $ax + by + c$ can be calculated with:

$$\mathrm{Var}(a \times x + b \times y + c) = a^2 \times \mathrm{Var}(x) + b^2 \times \mathrm{Var}(y)$$

or equivalently

$$\sigma^2_{a \times x + b \times y + c} = a^2 \times \sigma^2_x + b^2 \times \sigma^2_y$$

12.1.8 Exercise Properties of Means and Variances: Soup

There can be considerable variation between the salt concentration of home-made soup, when prepared by different persons (more or less water, different ingredients, etc.). Assuming that the salt concentration (in g per 100g of soup) follows approximately a Normal distribution with mean 1.4, and standard deviation of 0.3. Assume that a randomly selected person is preparing a pot of soup and you eat 250g of it.

Simulate a 1000 times the amount of salt in a bowl of soup of 250g prepared by a different person. Make a histogram of the results. Check whether the mean and variance of the simulated data series matches with what you would have expected based on the properties of means and variances.

12.2 Experimental Design

R can be helpful in different ways in the design of experiments [24].

As you may remember the *three R's* rule in experimental design:

- **R**andomization
- **R**eplication
- **R**educing noise factor (blocking, covariates)

You are going to look at some possibilities to create some specific randomized schemes: Completely Randomized Designs (CRD) and Randomized Complete Block Design (RCBD).

12.2.1 Package agricolae

The **agricolae** package contains some functions, that can help in forming specific experimental designs including the necessary randomization.

12.2.1.1 Completely Randomized Design (CRD)

In a CRD you let chance decide which treatment is allocated to which experimental unit. The function *design.crd()* needs a vector with treatment names and the number of replicates. Suppose you have a single treatment factor with 7 levels, and you want 4 replicates per treatment. So in total, there are $4 \times 7 = 28$ plots (experimental units) available, numbered as 1 to 28. Argument `seed` is just to start the random number generator. If you start with the same seed, you will get the same randomization scheme.

```
library(agricolae)
trt <- c("A", "B", "C", "D", "E", "F", "Ctrl")
r <- 4
seed <- 753
CRDdesign <- design.crd(trt = trt,
                        r = r,
                        serie = 0,
                        seed = seed)
CRDbook <- CRDdesign$book # field book
CRDbook[, c(1, 3)]
```

```
#>    plots  trt
#> 1      1    B
#> 2      2    F
#> 3      3 Ctrl
#> 4      4    A
#> 5      5    F
#> 6      6    D
#> 7      7    F
#> 8      8    C
#> 9      9    D
#> 10    10    C
#> 11    11    E
```

```
#> 12    12    B
#> 13    13    B
#> 14    14 Ctrl
#> 15    15    A
#> 16    16    A
#> 17    17    D
#> 18    18    C
#> 19    19    E
#> 20    20    C
#> 21    21    B
#> 22    22    D
#> 23    23 Ctrl
#> 24    24 Ctrl
#> 25    25    E
#> 26    26    F
#> 27    27    E
#> 28    28    A
```

12.2.1.2 Randomized Complete Block Design (RCBD)

Now comparable experimental units are grouped into blocks. Assuming that the block size is the same. Again you have 7 treatments, but now there should be 4 blocks. Within a block all 7 treatments should occur exactly once. Such blocks are called *complete*, because all treatment occur within the block.

```
trt <- c("A", "B", "C", "D", "E", "F", "Ctrl")
r <- 4  # Now reflecting the number of blocks
seed <- 123
RCBDdesign <- design.rcbd(trt = trt,
                          r = r,
                          serie = 0,
                          seed = seed)
(RCBDbook <- RCBDdesign$book) # Field book
```

```
#>    plots block  trt
#> 1     11     1 Ctrl
#> 2     12     1    A
#> 3     13     1    F
#> 4     14     1    D
#> 5     15     1    B
#> 6     16     1    E
#> 7     17     1    C
#> 8     21     2    D
#> 9     22     2    B
#> 10    23     2 Ctrl
#> 11    24     2    A
#> 12    25     2    F
#> 13    26     2    E
#> 14    27     2    C
#> 15    31     3    B
#> 16    32     3    A
#> 17    33     3    E
#> 18    34     3    D
#> 19    35     3    F
#> 20    36     3    C
#> 21    37     3 Ctrl
#> 22    41     4    B
#> 23    42     4    D
#> 24    43     4    A
#> 25    44     4    F
```

```
#> 26    45     4    C
#> 27    46     4    E
#> 28    47     4 Ctrl
```

Please notice the difference in the randomization schemes between the CRD and the RCBD.

12.2.1.3 Factorial design

In a factorial design there is more than one treatment factor, with crossed levels. The treatment combinations could be randomized according to a CRD or a RCBD.

```
trt <- c(3, 2) # factorial 3x2 (factor A: 3 levels, factor B: 2 levels),
# with 5 replicates in a CRD
factdesign1 <- design.ab(trt = trt,
                         r = 5,
                         serie = 2,
                         design = "crd")
(book1 <- factdesign1$book)
```

```
#>    plots r A B
#> 1    101 1 2 1
#> 2    102 2 2 1
#> 3    103 1 1 1
#> 4    104 1 3 2
#> 5    105 2 3 2
#> 6    106 3 2 1
#> 7    107 1 1 2
#> 8    108 1 2 2
#> 9    109 4 2 1
#> 10   110 2 1 1
#> 11   111 3 1 1
#> 12   112 1 3 1
#> 13   113 3 3 2
#> 14   114 2 1 2
#> 15   115 2 3 1
#> 16   116 4 1 1
#> 17   117 2 2 2
#> 18   118 3 3 1
#> 19   119 4 3 1
#> 20   120 3 1 2
#> 21   121 4 3 2
#> 22   122 3 2 2
#> 23   123 4 2 2
#> 24   124 4 1 2
#> 25   125 5 1 1
#> 26   126 5 1 2
#> 27   127 5 3 1
#> 28   128 5 2 1
#> 29   129 5 3 2
#> 30   130 5 2 2
```

```
trt <- c(3,2) # factorial 3x2 (factor A: 3 levels, factor B: 2 levels),
# with 5 replicates in a RCBD
factdesign2 <- design.ab(trt = trt,
                         r = 5,
                         serie = 2,
                         design = "rcbd")
(book2 <- factdesign2$book)
```

```
#>    plots block A B
```

```
#> 1    101     1 2 2
#> 2    102     1 1 1
#> 3    103     1 2 1
#> 4    104     1 1 2
#> 5    105     1 3 2
#> 6    106     1 3 1
#> 7    107     2 1 1
#> 8    108     2 3 2
#> 9    109     2 3 1
#> 10   110     2 1 2
#> 11   111     2 2 2
#> 12   112     2 2 1
#> 13   113     3 2 2
#> 14   114     3 3 2
#> 15   115     3 3 1
#> 16   116     3 2 1
#> 17   117     3 1 1
#> 18   118     3 1 2
#> 19   119     4 2 1
#> 20   120     4 1 2
#> 21   121     4 2 2
#> 22   122     4 3 1
#> 23   123     4 3 2
#> 24   124     4 1 1
#> 25   125     5 3 1
#> 26   126     5 3 2
#> 27   127     5 2 1
#> 28   128     5 2 2
#> 29   129     5 1 2
#> 30   130     5 1 1
```

12.2.2 Exercise Experimental Design: CRD and RCBD

Create a CRD and a RCBD for a single treatment factor with 5 levels, labeled A-E and 6 replicates (/blocks) per treatment.

12.3 Power Analysis and Sample Sizes

Power analysis, also known as Power Calculation, is an important aspect of experimental design. It allows you to determine the sample size required to detect an effect of a given size with a given degree of confidence. Conversely, it allows us to determine the probability of detecting an effect of a given size with a given level of confidence, under sample size constraints. If the probability is unacceptably low, it would be wise to alter or abandon the experiment.

The following four quantities have an intimate relationship:

1. sample size
2. effect size
3. significance level
4. power $= 1 - P(\text{Type II error}) = 1 - \beta$

Given any three, the fourth can be determined.

R has functions that can be helpful in deciding how large a study should be so that specific requirements are satisfied. These functions are specific for the statistical test you have in mind. For instance, for *t*-tests the function *power.t.test()* from the **stats** package can be used.

Without going into statistical details now, the function *power.t.test()* needs four out of five arguments to be specified, and then the fifth argument will be calculated from the others. These five arguments are:

1. n = sample size
2. Δ = delta = difference of interest
3. sd = standard deviation
4. sig.level = α = significance level or $P(\text{Type I error})$, is the probability of finding an effect that is not there. In order words the chance, that is maximally accepted, of erroneously rejecting H_0.
5. power = $1 - P(\text{Type II error}) = 1 - \beta$ = probability to reject H_0 if the differences of means is equal to delta. In order words this is probability of finding an effect that is there, or correctly rejecting H_0 and accepting H_a.

Besides these arguments you need to specify the type of t-test (two independent samples, single sample, paired samples) and the alternative hypothesis (two-sided or one-sided).

Below an example is shown of calculating the sample size for a one-sample t-test. The difference of interest is 1, the standard deviation is also 1 and the power should be 0.90 if testing at significance level 0.01.

```
outcome <- power.t.test(
  delta = 1,                # difference of interest
  sd = 1,                   # standard deviation
  sig.level = 0.01,         # significance level
  power = 0.90,             # power
  type = "one.sample",      # type of study
  alternative = "two.sided" # type of test
  )
print(outcome)
```

```
#>
#>      One-sample t test power calculation
#>
#>               n = 18.30346
#>           delta = 1
#>              sd = 1
#>       sig.level = 0.01
#>           power = 0.9
#>     alternative = two.sided
```

The calculated sample size yields 19 (rounded upwards from 18.3).

In the next example the power of the two-sided one sample t-test is calculated, when the sample size would be equal to 20.

```
power.t.test(n = 20,
             delta = 1,
             sd = 1,
             sig.level = 0.01,
             type = "one.sample",
             alternative = "two.sided"
             )
```

```
#>
#>      One-sample t test power calculation
#>
#>               n = 20
#>           delta = 1
#>              sd = 1
#>       sig.level = 0.01
#>           power = 0.9324954
#>     alternative = two.sided
```

12.3.1 The pwr Package

The **pwr** package developed by Stéphane Champely, implements power analysis as outlined by Cohen [25]. Some of the more important functions are listed in Table 12.2.

Table 12.2: Important functions from the **pwr** package.

Function	Power calculations for
pwr.2p.test()	two proportions (equal n)
pwr.2p2n.test()	two proportions (unequal n)
pwr.anova.test()	balanced one way ANOVA
pwr.chisq.test()	chi-square test
pwr.f2.test()	general linear model
pwr.p.test()	proportion (one sample)
pwr.r.test()	correlation
pwr.t.test()	t-tests (one sample, 2 samples, paired)
pwr.t2n.test()	t-test (two samples with unequal n)

For each of these functions, you enter three of the four quantities (effect size, sample size, significance level, power) and the fourth is calculated.

The significance level defaults to 0.05. Therefore, to calculate the significance level, given an effect size, sample size, and power, use `sig.level = NULL` as function argument.

Specifying an effect size can be a daunting task. Effect size formulas and Cohen's suggestions (based on social science research) are provided below. Cohen's suggestions should only be seen as very rough guidelines. Your own subject matter experience should be brought to bear.

12.3.1.1 *t*-tests

For t-tests, use the following functions:

```
pwr.t.test(n = , d = , sig.level = , power = ,
           type = c("two.sample", "one.sample", "paired"))
```

where `n` is the sample size, `d` is the effect size, and type indicates a two-sample t-test, one-sample t-test or paired t-test. If you have unequal sample sizes, use

```
pwr.t2n.test(n1 = , n2 = , d = , sig.level = , power = )
```

where `n1` and `n2` are the sample sizes of the two.

For t-tests, the effect size is assessed as

$$d = \frac{|\mu_1 - \mu_2|}{\sigma}$$

with μ_1 = mean of group 1, μ_2 = mean of group 2, and σ^2 = common error variance.

Cohen suggests that `d` values of 0.2, 0.5, and 0.8 represent small, medium, and large effect sizes respectively.

You can specify `alternative = "two.sided"`, `"less"`, or `"greater"` to indicate a two-tailed, or one-tailed test. A two-tailed test is the default.

12.3.1.2 ANOVA

For a one-way analysis of variance use

```
pwr.anova.test(k = , n = , f = , sig.level = , power = )
```

where `k` is the number of groups and `n` is the common sample size in each group.

For a one-way ANOVA effect size is measured by `f` where

$$f = \sqrt{\frac{\sum_{i=1}^{k} p_i \times (\mu_i - \mu)^2}{\sigma^2}}$$

with $p_i = n_i/N$, n_i = number of observations in group i, N = total number of observations, μ_i = mean of group i, μ = grand mean, σ^2 = error variance within groups.

Cohen suggests that `f` values of 0.1, 0.25, and 0.4 represent small, medium, and large effect sizes respectively.

12.3.1.3 Correlations

For correlation coefficients use

```
pwr.r.test(n = , r = , sig.level = , power = )
```

where `n` is the sample size and `r` is the correlation. You use the population correlation coefficient as the effect size measure. Cohen suggests that `r` values of 0.1, 0.3, and 0.5 represent small, medium, and large effect sizes respectively.

12.3.1.4 Linear Models

For linear models (*e.g.*, multiple linear regression) use

```
pwr.f2.test(u = , v = , f2 = , sig.level = , power = )
```

where `u` and `v` are the numerator and denominator degrees of freedom. You use `f2` as the effect size measure.

$$f^2 = \frac{R^2}{1 - R^2}$$

where R^2 = population squared multiple correlation.

$$f^2 = \frac{R^2_{AB} - R^2_A}{1 - R^2_{AB}}$$

where R^2_A = variance accounted for in the population by variable set A, R^2_{AB} = variance accounted for in the population by variable set A and B together.

The first formula is appropriate when you are evaluating the impact of a set of predictors on an outcome. The second formula is appropriate when you are evaluating the impact of one set of predictors above and beyond a second set of predictors (or covariates). Cohen suggests `f2` values of 0.02, 0.15, and 0.35 represent small, medium, and large effect sizes.

12.3.1.5 Tests of Proportions

When comparing two proportions use

```
pwr.2p.test(h = , n = , sig.level = , power = )
```

where `h` is the effect size and `n` is the common sample size in each group.

$$h = 2\arcsin\left(\sqrt{p_1}\right) - 2\arcsin\left(\sqrt{p_2}\right)$$

Cohen suggests that `h` values of 0.2, 0.5, and 0.8 represent small, medium, and large effect sizes respectively.

For unequal sample sizes use

```
pwr.2p2n.test(h = , n1 = , n2 = , sig.level = , power = )
```

To test a single proportion use

```
pwr.p.test(h = , n = , sig.level = power = )
```

For both two sample and one sample proportion tests, you can specify `alternative = "two.sided"`, `"less"`, or `"greater"` to indicate a two-tailed, or one-tailed test. A two tailed test is the default.

12.3.1.6 Chi-squared (χ^2) Tests

For chi-squared tests use

```
pwr.chisq.test(w = , N = , df = , sig.level = , power = )
```

where `w` is the effect size, `N` is the total sample size, and `df` is the degrees of freedom. The effect size `w` is defined as

$$w = \sqrt{\sum_{i=1}^{m} \frac{(p0_i - pa_i)^2}{p0_i}}$$

where $p0_i$ = cell probability in the i^{th} cell under H_0, pa_i = cell probability in the i^{th} cell under H_a.

Cohen suggests that `w` values of 0.1, 0.3, and 0.5 represent small, medium, and large effect sizes respectively.

12.3.2 Exercise Power Analysis: one-sided t-test

Calculate sample sizes with *power.t.test()* for the following situations:

- Independent-samples t-test, one-sided alternative hypothesis; power of test equal to 0.80 to find a difference between means in the two samples equal to 10 with standard deviation equal to 3, and significance level $\alpha = 0.05$.
- Paired samples t-test, same other arguments.
- Independent-samples t-test, two-sided alternative hypothesis, same other arguments.

12.3.3 Exercise Power Analysis: a two-sided independent-samples t-test

Calculate the power for an independent-samples two-sided t-test, using $\alpha = 0.05$, sample size $n = 20$, and standard deviation $= 10$, for a range of differences delta (from 0 to 25 in small steps, e.g. 0.1) using the *power.t.test()* function. Argument `delta` may be a vector, and the result will be a power vector. The power may be extracted from the resulting object as a component from a list. Plot the resulting power (on the y-axis) versus the delta (on the x-axis).

Add lines into the graph for sample sizes 10 and 5.

12.4 Command Reference

Table 12.3: Lesson 12 commands

Command	Description
design.crd()	Completely Randomized Design from **agricolae** package
power.t.test()	power calculations for one and two sample t tests
sample()	random samples and permutations

Lesson 13

Graphics with the ggplot2 and lattice packages

13.1 Graphics with the ggplot2 package

This text is largely taken, with permission, from the reader of the Graduate School for Production Ecology and Resource Conservation (PE&RC) PhD course "Introduction to R for statistical analysis" by John Bastiaansen, Albart Coster and Gerrit Gort.

In case you would like to learn more about the **ggplot2** package, go to the following web address https://dx.doi.org/10.1007/978-3-319-24277-4, while being connected to the Wageningen University Wireless network named Eduroam, and download the book "ggplot2:Elegant Graphics for Data Analysis" as pdf- or epub-file for free.

13.1.1 The ggplot2 package

The **ggplot2** [20] package is based on the *grammar of graphics*. It is important to mention, because it has become very popular among R Users. Although the function calls in the **ggplot2** package do not follow the default R language, it is very useful to create a wide variety of graphics in R.

Users who wish to create a graphic using the **ggplot2** package need to think about the following aspects:

- The *data* and the *aesthetic mapping* of the data.
- Geometric objects, *geoms*, represent what you see on the plot (points, lines, polygons).
- Statistical transformations of the data, *stats*, summarize and transform the data before displaying them in the plot.
- The *scales* map the (transformed) values in the data to an aesthetic space. The scale also provides the legend.
- A coordinate system, *coord*.
- A *facetting* specification describes how to break up the data into subsets.

13.1.2 Creating a plot

You use the *ggplot()* function to create a plot object. The *ggplot()* function has two arguments: `data` and aesthetic mapping, which set up the defaults of the plot object and can also be specified in each layer. The aesthetic mapping of the data can be specified with the *aes()* function.

```
p <- ggplot(data = ChickWeight, aes(x = Time, y = weight))
```

This plot cannot be displayed until we add a layer, since there is nothing to see.

13.1.3 Adding layers to the plot

To display something, you need to add a layer to the plot. The layers can be added with the *layer()* function:

```
layer(geom, stat, data, mapping, position, params, inherit.aes,
      check.aes, check.param. show.legend, key_glyph, layer_class)
```

Or you can use specific functions to add specific layers:

- *geom_bar()* to add bars to the plot.
- *geom_errorbar()* to add error bars.
- *geom_point()* to add points to the plot.
- *geom_line()* to add lines.
- *geom_histogram()* to add a histogram.

You can also perform some analyses before and subsequently add a layer to the plot:

- *stat_summary()* to summarize the data at each x-value.
- *stat_smooth()* to draw a smoothed line through the data.

13.1.4 Adding points

Now, with the following command you add points representing the individual weights to the plot (Figure 13.1):

```
p + geom_point()
```

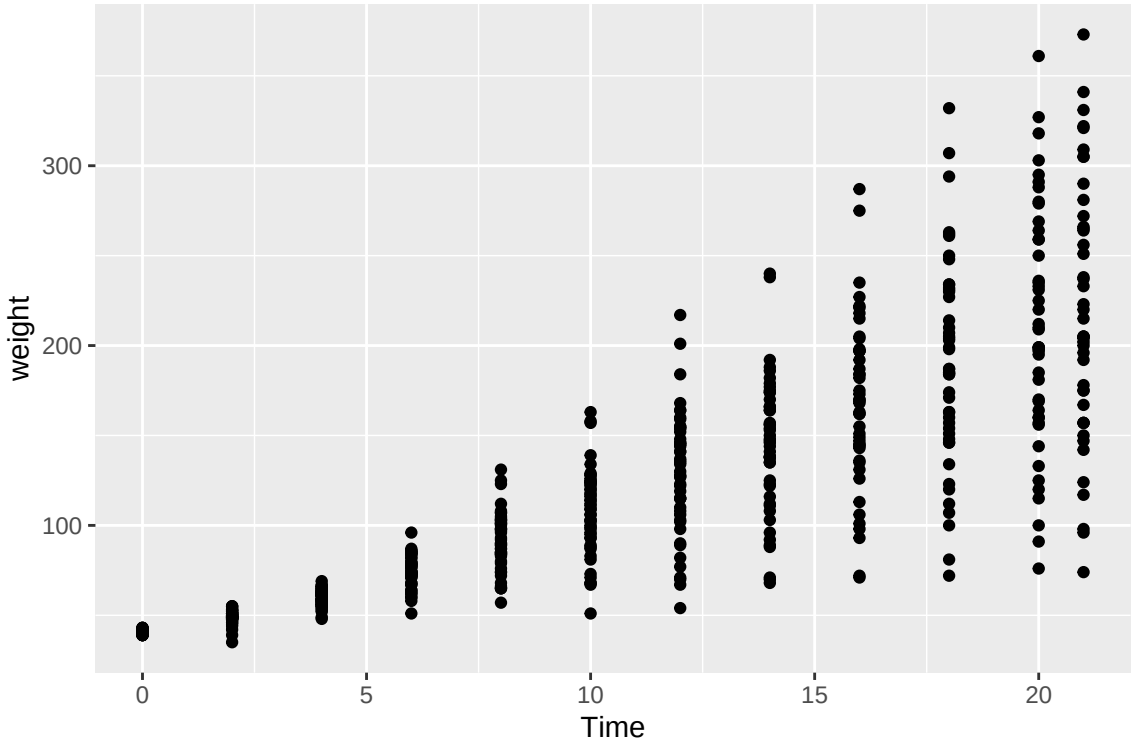


Figure 13.1: Scatterplot of chick weights plotted against time.

13.1.5 Differentiating between chicks

You can differentiate between the chicks using distinct colors for each chick (Figure 13.2):

```
p + geom_point(aes(colour = Chick))
```

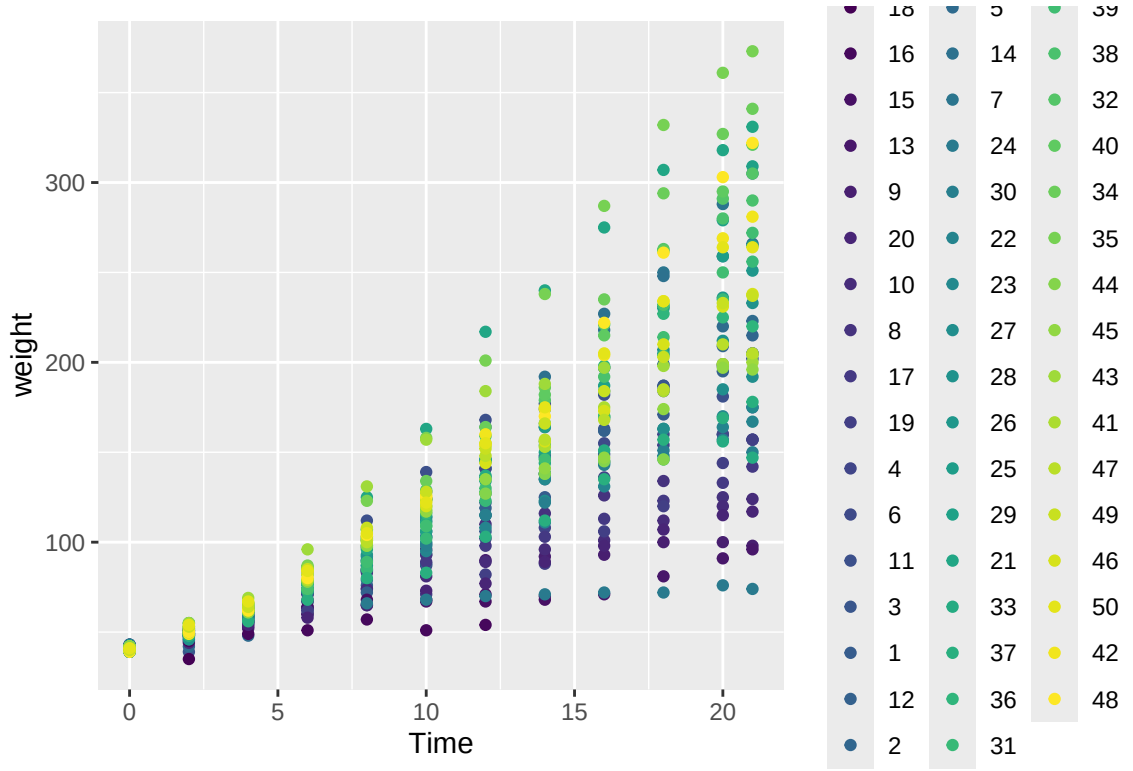


Figure 13.2: Scatterplot of differentiated chick weights plotted against time.

Since the number of chicks is so large, the legend is useless. You can use the *theme()* function to remove the legend (Figure 13.3):

```
p + geom_point(aes(colour = Chick)) + theme(legend.position = "none")
```

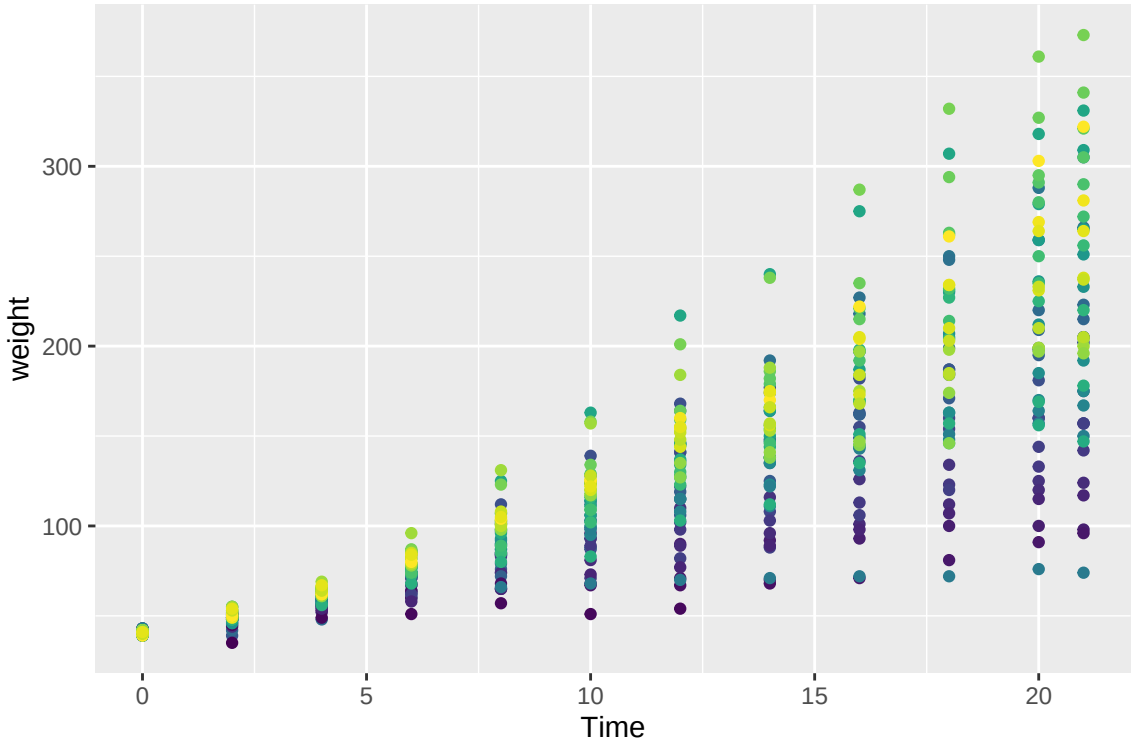


Figure 13.3: Scatterplot of differentiated chick weights plotted against time without legend.

You can also use a distinct shape for each chick (Figure 13.4):

```
p + geom_point(aes(shape = Chick, colour = Chick)) + theme(legend.position = "none")
# Warning messages:
# 1: Using shapes for an ordinal variable is not advised
# 2: The shape palette can deal with a maximum of 6 discrete values because more
#    than 6 becomes difficult to discriminate; you have 50. Consider specifying
#    shapes manually if you must have them.
# 3: Removed 525 rows containing missing values (`geom_point()`).
```

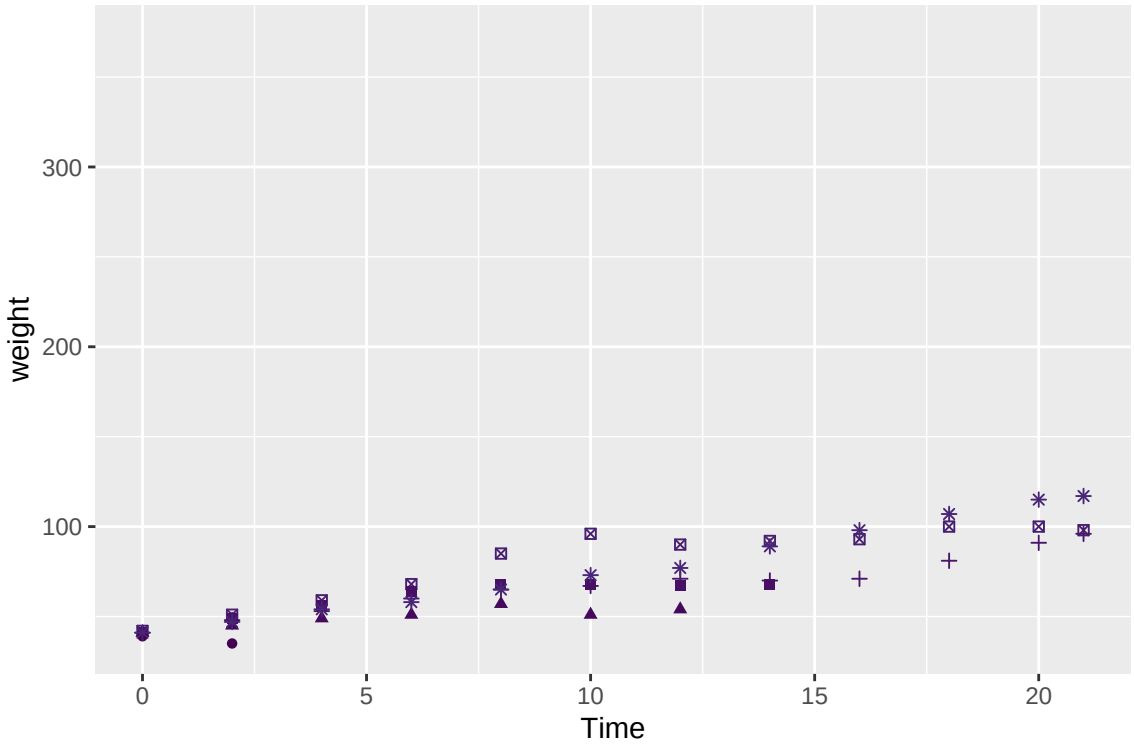


Figure 13.4: Scatterplot of differentiated chick weights, with distinct shapes, plotted against time.

Since there are more than 6 Chicks, you should specify the shapes manually. First, let's have a look at the different point shapes available in the **ggplot2** package (Figure 13.5):

```
df <- data.frame(x = 1:12, y = 1, shapes = as.character(1:6), colours =  as.character(1:6))
ggplot(data = df, aes(x = x, y = y, colour = colours, shape = shapes)) + geom_point()
```

Now let's apply these different shapes and colors (Figure 13.6):

```
ChickWeight$shape <- factor(as.numeric(ChickWeight$Chick) %% 6)
p$data <- ChickWeight
p + geom_point(aes(shape = shape, colour = Chick)) + theme(legend.position = "none")
```

The modulo operator %% of R is used above. The modulo operation finds the remainder of the division of one number by another.

13.1.6 Connecting the chicks by a line

Since there are multiple observations on each chick, you can connect the observations by individual lines. For this you need to add another layer using the *geom_line()* function. Simply adding `+ geom_line()` does not work correctly, since **ggplot2** does not know which observations should be connected to each other as shown in Figure 13.7.

```
p + geom_line()
```

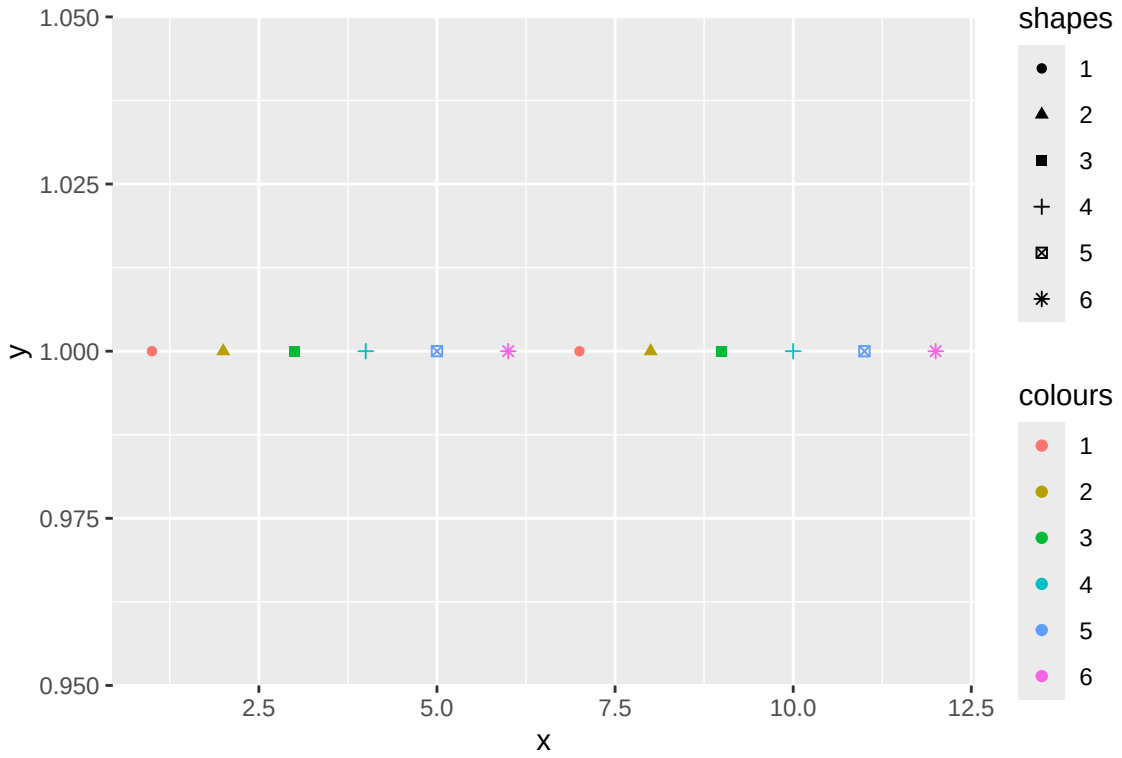


Figure 13.5: Scatterplot of different shapes in ggplot2.

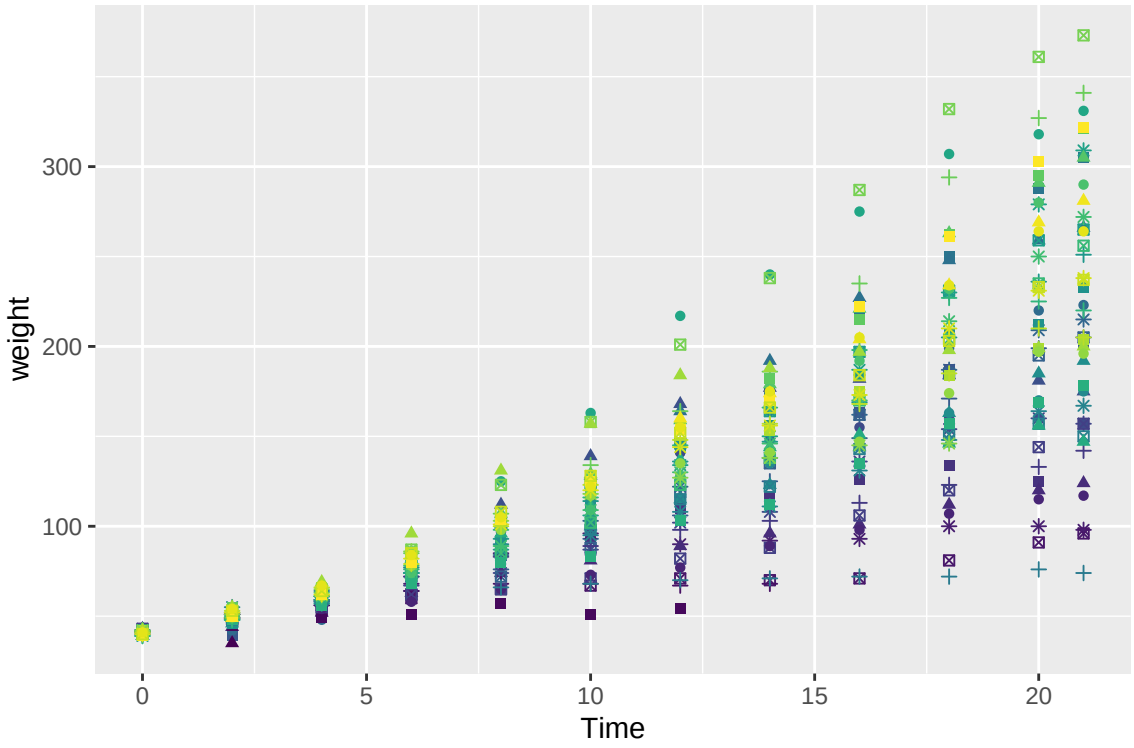


Figure 13.6: Scatterplot of differentiated chick weights, with distinct shapes and colours, plotted against time.

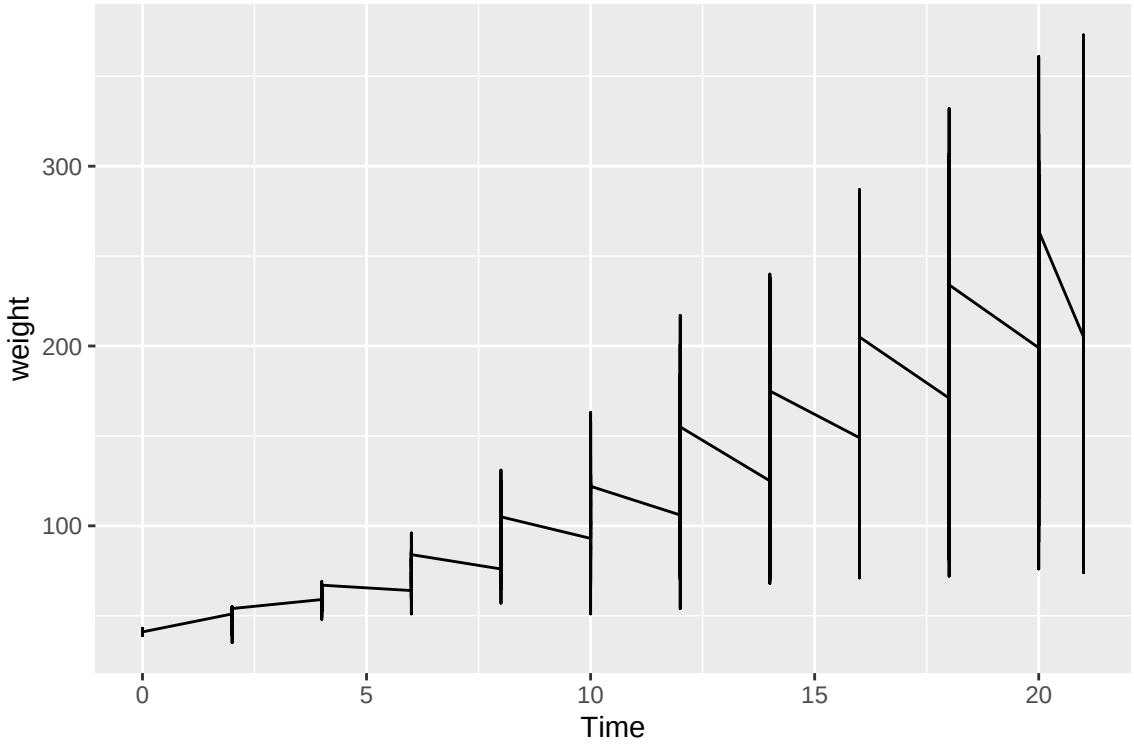


Figure 13.7: Incorrect connection of chicks over time by simply adding *geom line()* to a ggplot.

Hence, you need to tell that each chick over time is a group. Furthermore, you can specify the type of the lines and the colour of the lines as done previously for the points (Figure 13.8):

```
p <- ggplot(data = ChickWeight,
            aes(x = Time,
                y = weight,
                colour = shape,
                group = Chick,
                linetype = shape))
p + geom_line() + theme(legend.position = "none")
```

13.1.7 Summarizing data

You might want to summarize the observations in a graphic as depicted in Figure 13.9:

```
ChickWeight$Diet <- factor(ChickWeight$Diet)
p <- ggplot(data = ChickWeight, aes(x = Time, y = weight))
p + stat_summary(fun = "mean",
                 geom = "line",
                 aes(group = Diet, colour = Diet)) +
    stat_summary(fun = "sd",
                 geom = "point",
                 aes(group = Diet, colour = Diet))
```

In another plot (Figure 13.10) you might also want to show the range of the data at each point (lines show the 5 and 95 percent quantiles, points showing the median):

```
q5  <- function(x) quantile(x, probs = 0.05)
q95 <- function(x) quantile(x, probs = 0.95)
p + stat_summary(fun = "median",
                 fun.min = q5,
                 fun.max = q95,
                 aes(group = Diet, colour = Diet))
```

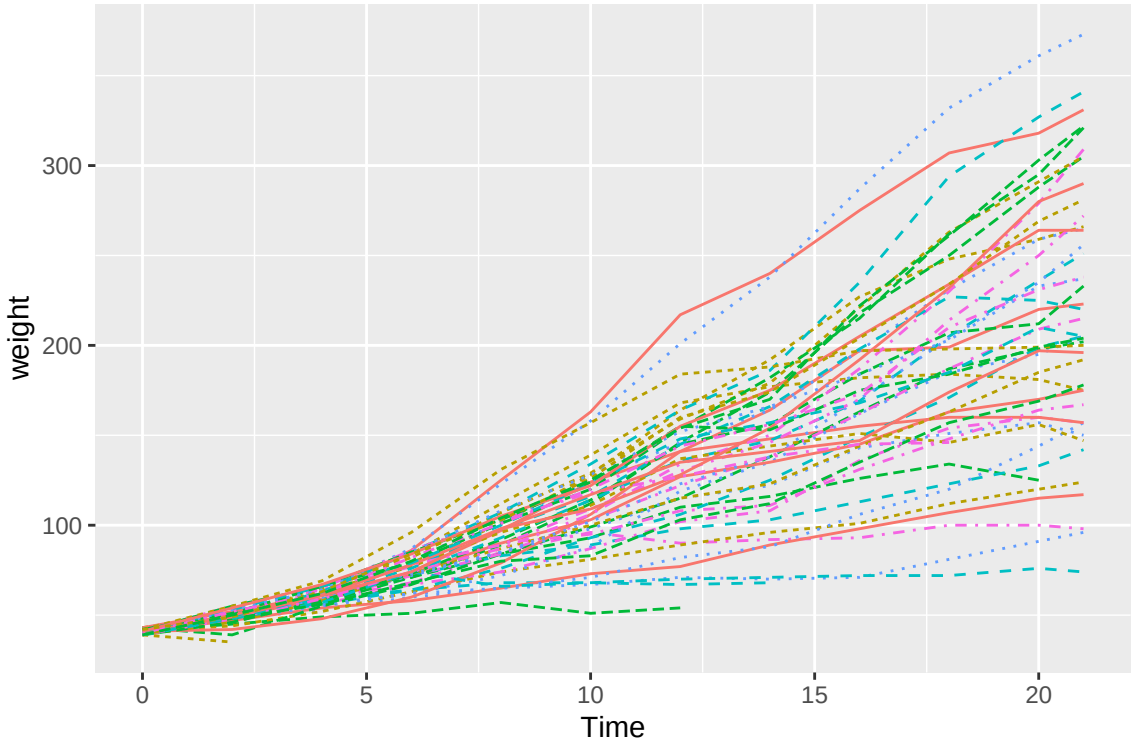


Figure 13.8: Scatterplot of differentiated chick weights, with distinct lines and colours, plotted against time.

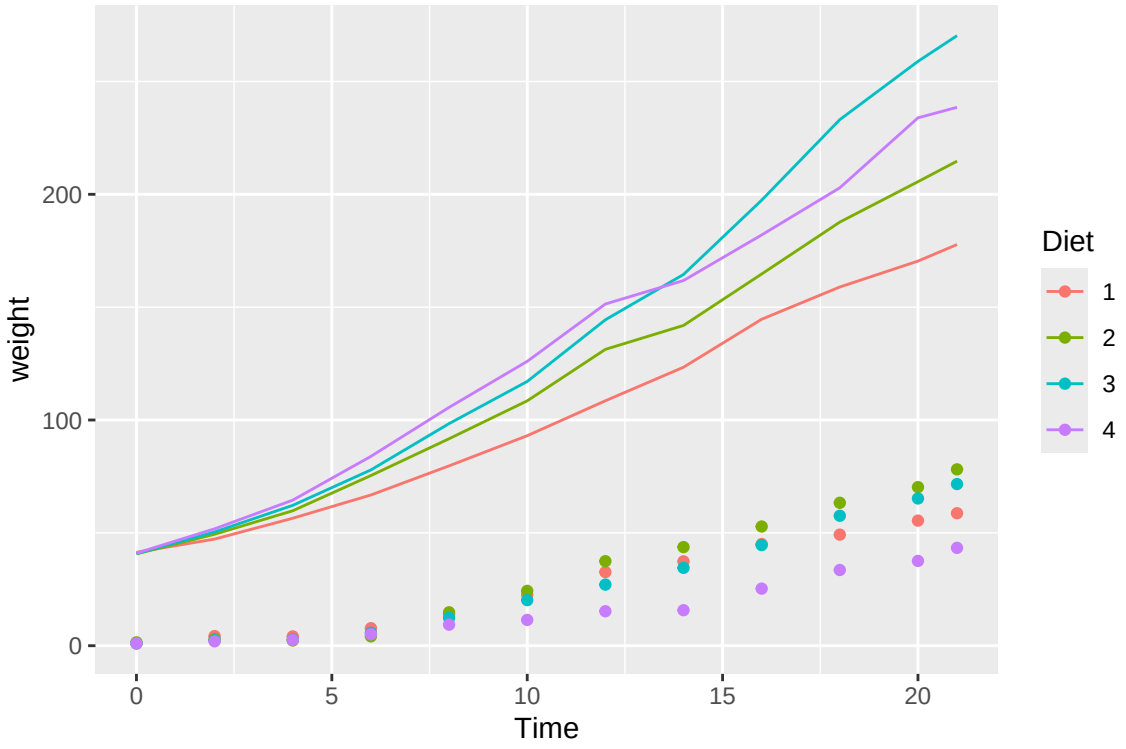


Figure 13.9: Plot summarizing the mean (line) and standard deviation (dot) per diet over time.

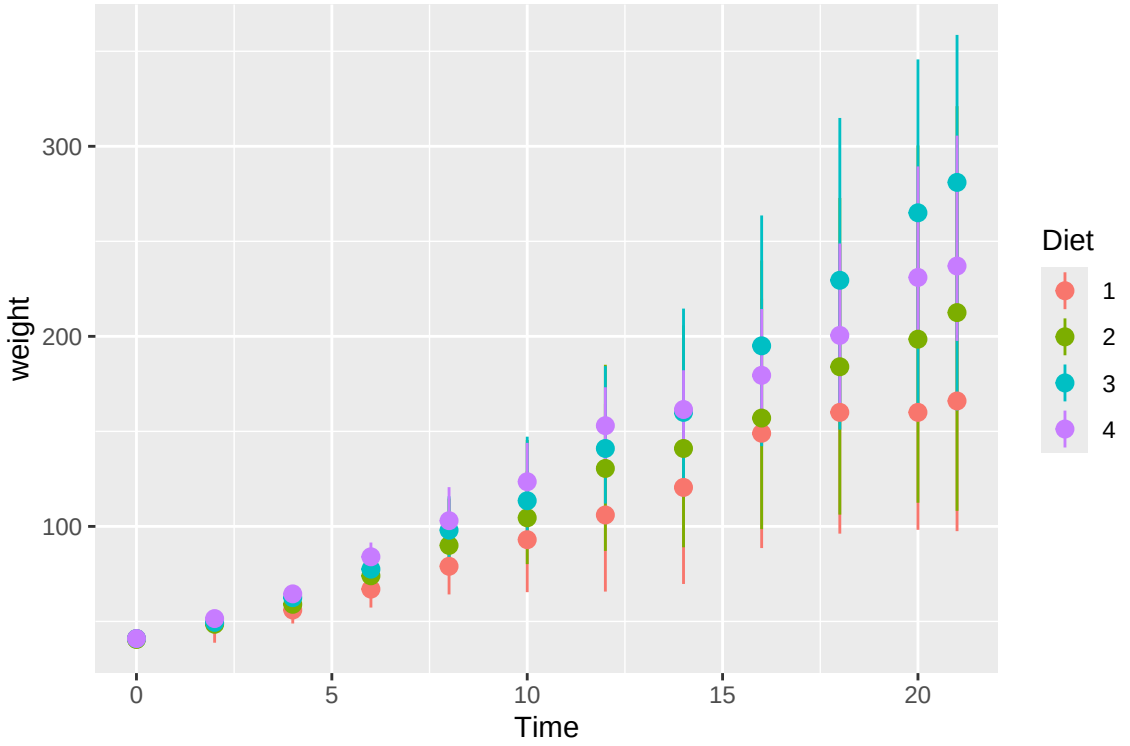


Figure 13.10: Plot summarizing the median (dot) and 5 and 95 percent quantiles (line) per diet over time.

As you can see in Figure 13.10, this is not very useful since the lines overlap each other. There are several alternatives. The first is to **jitter** the points to avoid overlapping as shown in Figure 13.11:

```
q5  <- function(x) quantile(x, probs = 0.05)
q95 <- function(x) quantile(x, probs = 0.95)
p + stat_summary(fun = "median",
                 fun.min = q5,
                 fun.max = q95,
                 position = "jitter",
                 aes(group = Diet, colour = Diet))
```

A second option is to make a distinct plot for each diet, using a technique called *faceting*. Now the colors are no longer needed to identify the diets and you can use a single color as depicted in Figure 13.12:

```
levels(p$data$Diet) <- paste("Diet", levels(p$data$Diet))
p + stat_summary(fun = "median",
                 fun.min = q5,
                 fun.max = q95,
                 colour = "red",
                 aes(group = Diet)) +
    facet_grid(. ~ Diet)
```

13.1.8 Barplots

To make a barplot, there are two alternatives. First, you can provide summarized data, for example the average weight per diet and per point in time:

```
sw <- with(data = ChickWeight,
           tapply(X = weight,
                  INDEX = list(Time, Diet),
                  FUN = mean,
                  na.rm = TRUE))
head(sw)
```

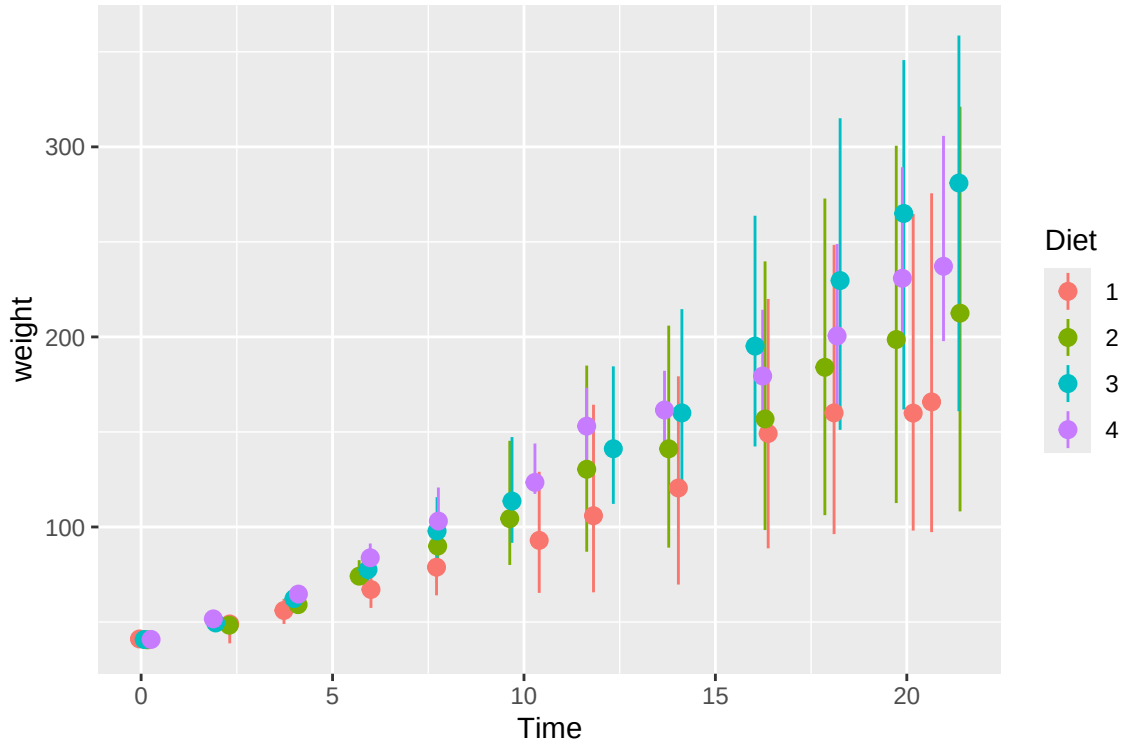


Figure 13.11: Plot summarizing the median (dot) and 5 and 95 percent quantiles (line) per diet over time displayed with jittering.

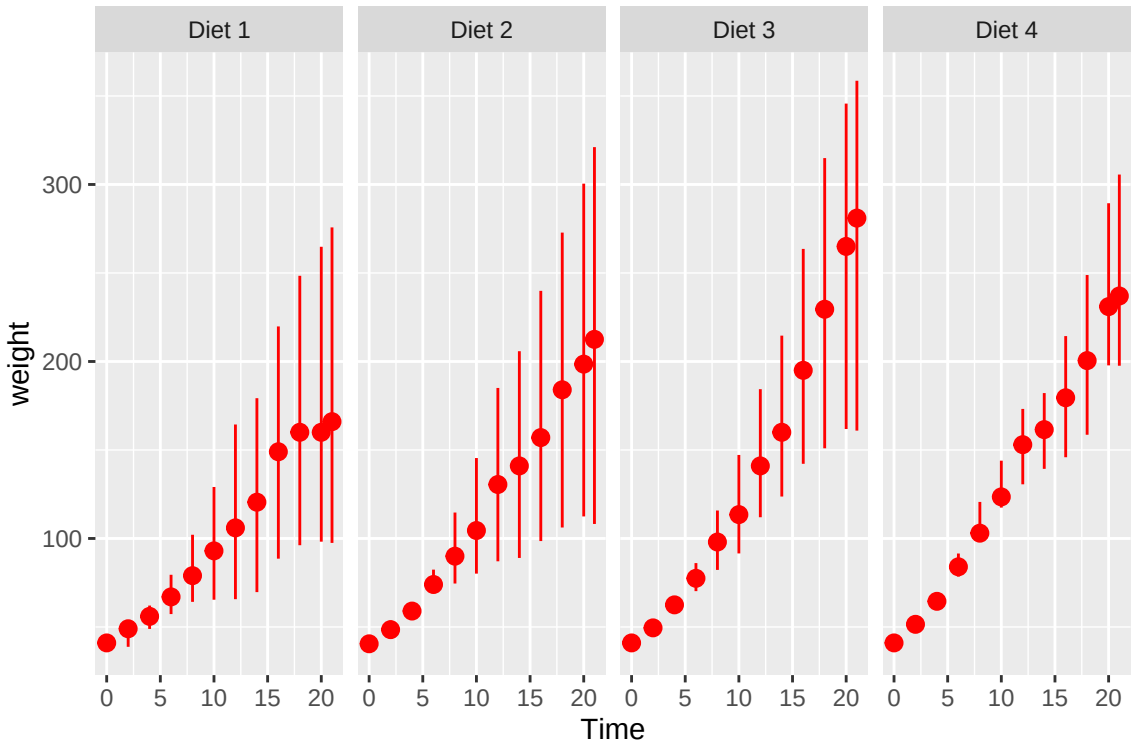


Figure 13.12: Plot summarizing the median (dot) and 5 and 95 percent quantiles (line) per diet over time with faceting.

```
#>           1     2     3     4
#> 0  41.40000  40.7  40.8  41.0
#> 2  47.25000  49.4  50.4  51.8
#> 4  56.47368  59.8  62.2  64.5
#> 6  66.78947  75.4  77.9  83.9
#> 8  79.68421  91.7  98.4 105.6
#> 10 93.05263 108.5 117.1 126.0
```

Object *sw* has a format, which is not usable for the **ggplot2** package. Therefore. you need to have the data in *long* format using the *melt()* function in the **reshape2** package:

```
swl <- reshape2::melt(data = sw, value.name = "weight")
head(swl)
```

```
#>   Var1 Var2   weight
#> 1    0    1 41.40000
#> 2    2    1 47.25000
#> 3    4    1 56.47368
#> 4    6    1 66.78947
#> 5    8    1 79.68421
#> 6   10    1 93.05263
```

Renaming the other two columns with proper variable names:

```
swl <- dplyr::rename(swl, Time = Var1, Diet = Var2)
swl$Diet <- factor(swl$Diet)
```

Now the barplot can be created as displayed in Figure 13.13.

```
bp <- ggplot(data = swl, aes(y = weight, x = Time, fill = Diet))
bp + geom_bar(stat =  "identity")
```

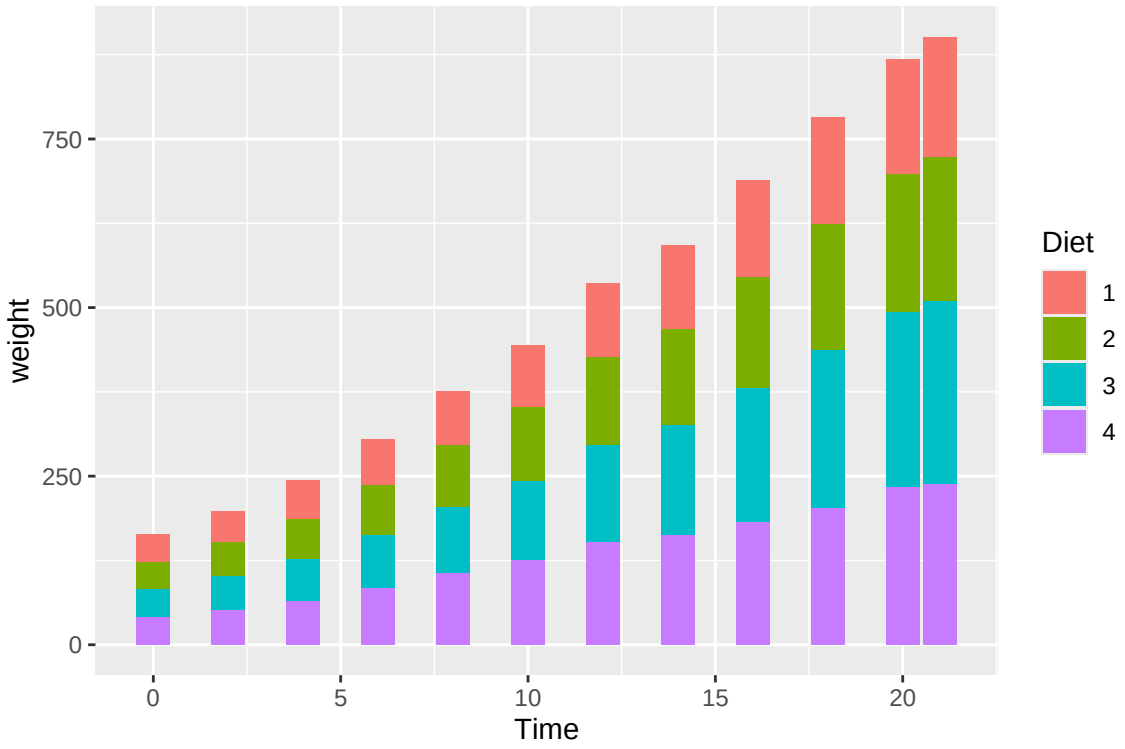


Figure 13.13: Barplot of the average weight per diet and per point in time with stacked bars.

By default, the bars are stacked upon each other. However, you can also place them next to each other as shown in Figure 13.14:

```
bp + geom_bar(stat =  "identity", position = "dodge")
```

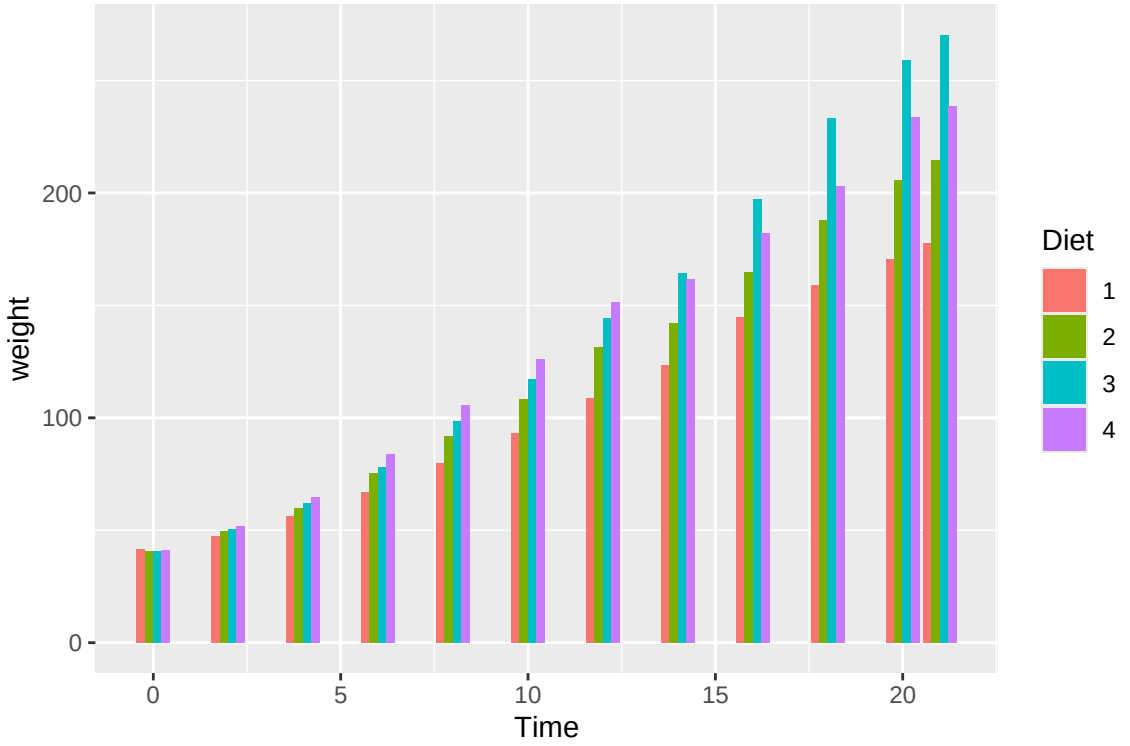


Figure 13.14: Barplot of the average weight per diet and per point in time with bars side-by-side.

Instead of calculating summary statistics beforehand, you can also use the *stat_summary()* function to calculate the average weight per point in time, as shown previously. Argument `position = "dodge"` specifies that the bars should be located next to each other. In case of using the function argument `position = "dodge2"` extra space will be added between the bars located next to each other. With the function argument `position = "stack"` the bars are placed on top of each other (stacked).

```
bp1 <- ggplot(data = ChickWeight, aes(y = weight, x = Time, fill = Diet))
```

The figures resulting from the shown commands look the same as shown, in respectively, Figure 13.13 and Figure 13.14.

```
bp1 + stat_summary(fun = mean, geom = "bar", position = "stack")
bp1 + stat_summary(fun = mean, geom = "bar", position = "dodge")
```

Since you are use the raw data, it should be possible to add bars showing the range of the data. For this, you use the *geom_errorbar()* function, as visually depicted in Figure 13.15:

```
bp1 + stat_summary(fun = mean,
                   geom = "bar",
                   position = "dodge") +
      stat_summary(fun.max = q95,
                   fun.min = q5,
                   geom = "errorbar",
                   width = 0.1,
                   position = position_dodge(width = 0.9),
                   linewidth = 0.5)
```

13.1.9 Trendlines

Suppose you want to draw a smooth line through the data. For this you use the *stat_smooth()* function (Figure 13.16):

```
p <- ggplot(data = ChickWeight, aes(y = weight, x = Time))
p + geom_point(aes(colour = Diet)) +
    stat_smooth(method = "loess",
```

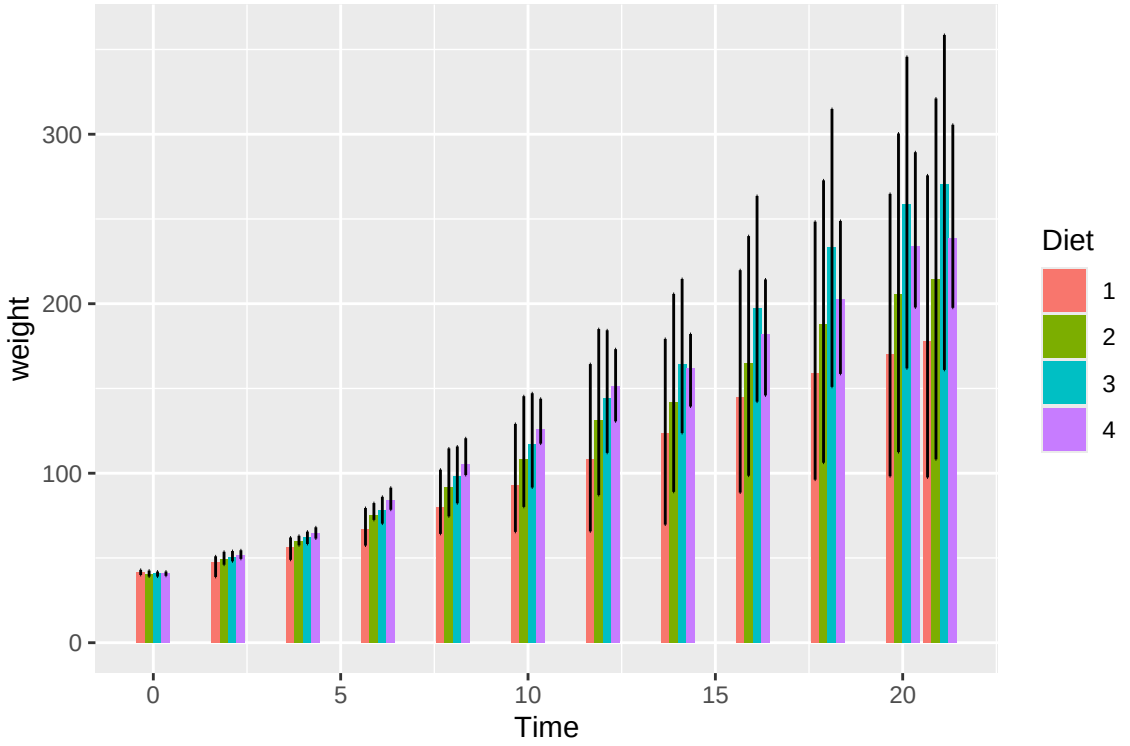


Figure 13.15: Side-by-side barplots of the average weight per diet and per point in time with errorbars indicating the range of the data.

```
            formula = y ~ x,
            se = FALSE)
```

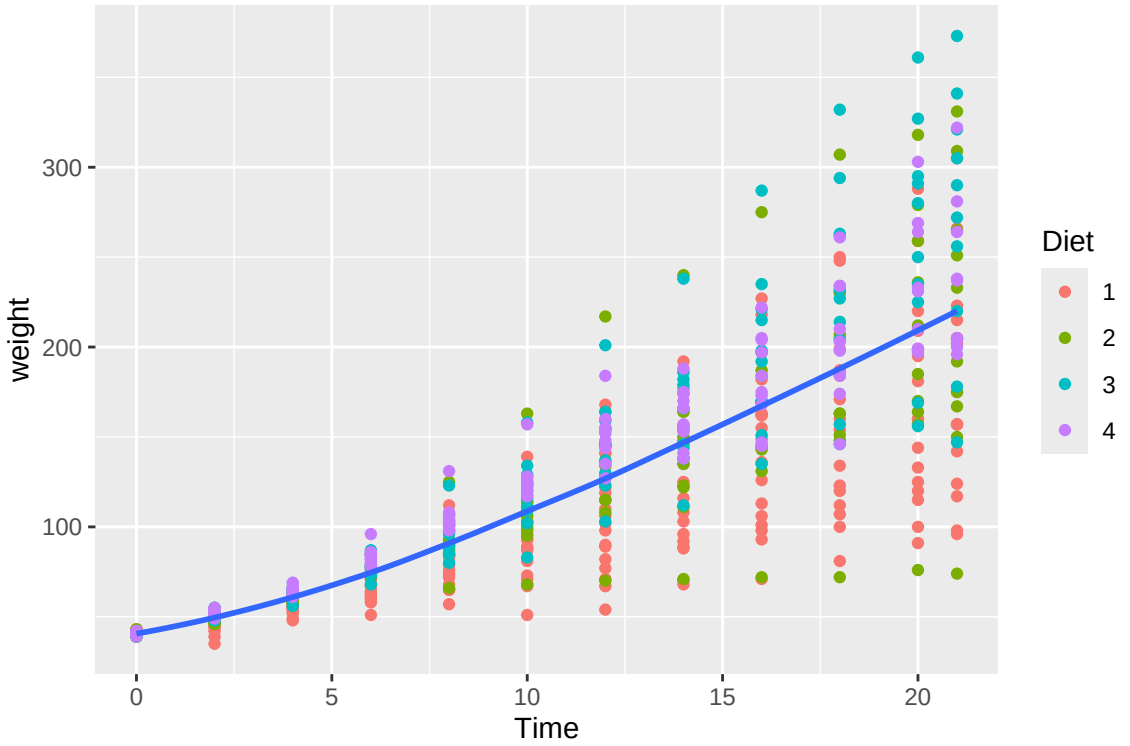


Figure 13.16: Scatterplot of chick weights, coloured by diet, plotted against time with a smooth line through all data points.

Smooth lines can also be added for each Diet separately as displayed in Figure 13.17:

```
p + geom_point(aes(colour = Diet)) +
    stat_smooth(method = "loess",
                formula = y ~ x,
                se = FALSE,
                colour = "black",
                linewidth = 1) +
    stat_smooth(method = "loess",
                formula = y ~ x,
                se = FALSE,
                aes(colour = Diet),
                linewidth = 0.8)
```

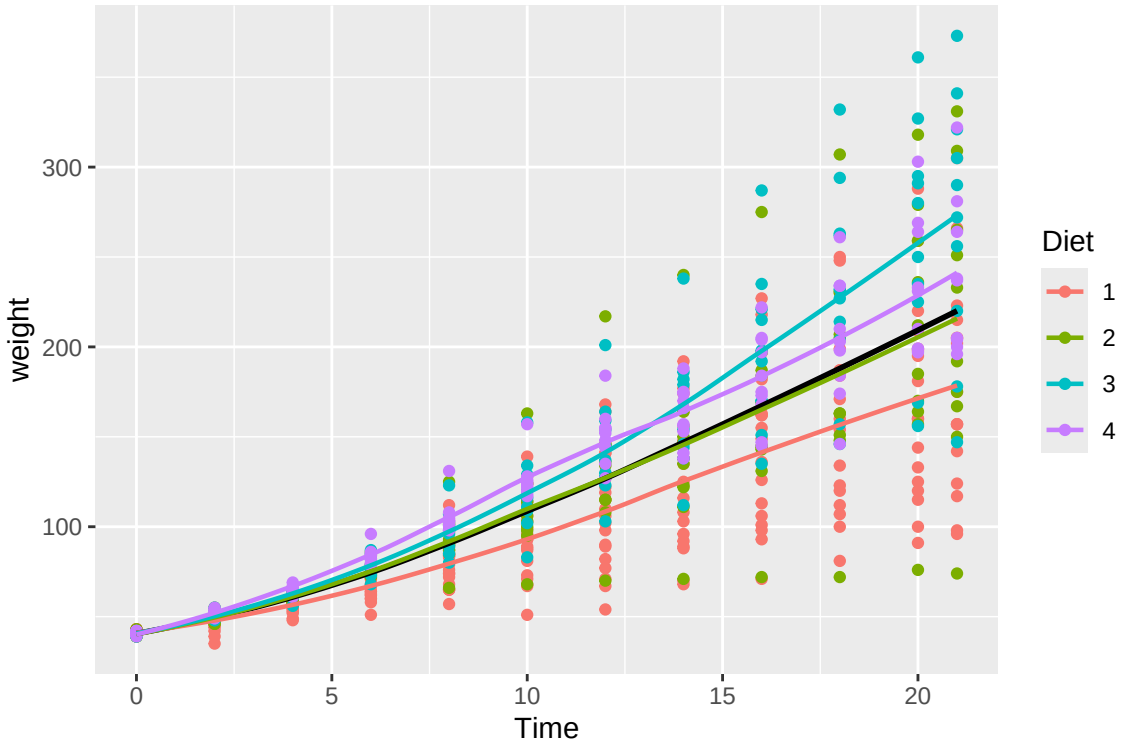


Figure 13.17: Scatterplot of chick weights, coloured by diet, plotted against time with smooth lines for each diet as well as through all data points.

In the previous example, the *stat_smooth()* function used a non-parametric method, called **lo**cal reg**ress**ion (`"loess"`), to fit the data. Linear regression can also be used, as shown in Figure 13.18, by running the following commands:

```
p + geom_point(aes(colour = Diet)) +
    stat_smooth(aes(colour = Diet),
                method = "lm",
                formula = y ~ x,
                linewidth = 1,
                se = FALSE)
```

As visually displayed in Figure 13.19, even the equation to use can be specified:

```
p + geom_point(aes(colour = Diet)) +
    stat_smooth(aes(colour = Diet),
                method = "lm",
                formula =  y ~ x + I(x^2),
                linewidth = 1,
                se = FALSE)
```

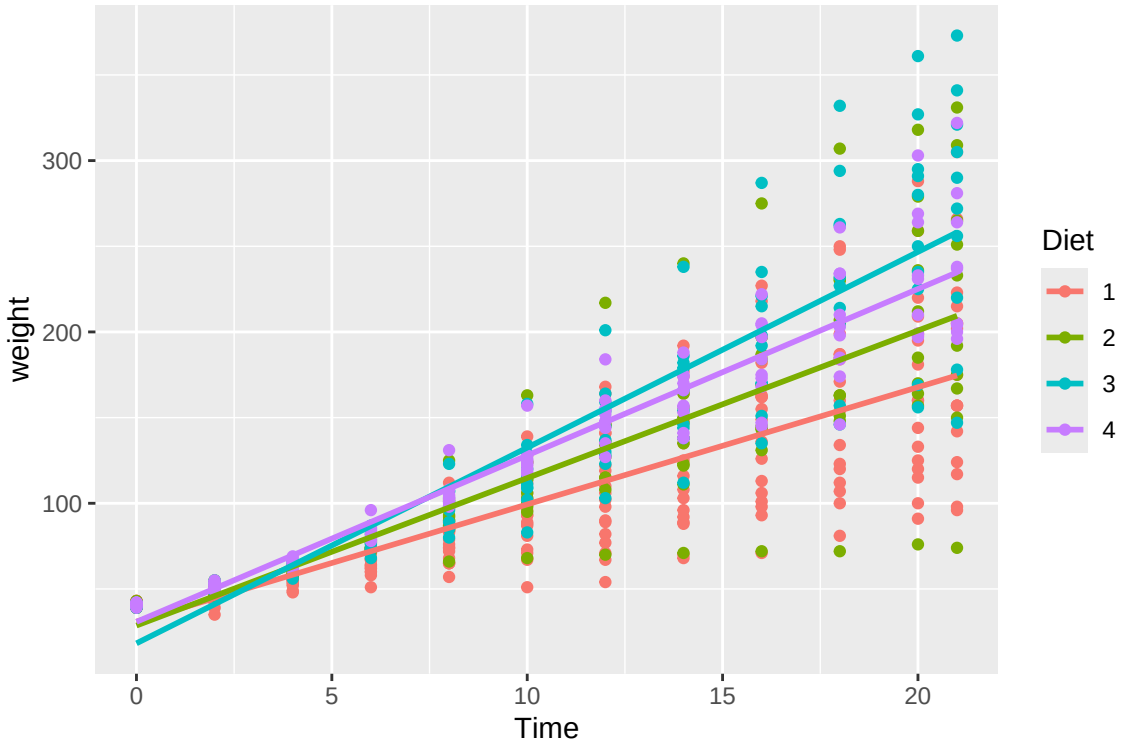


Figure 13.18: Scatterplot of chick weights, coloured by diet, plotted against time with first order linear least squares regression lines for each diet.

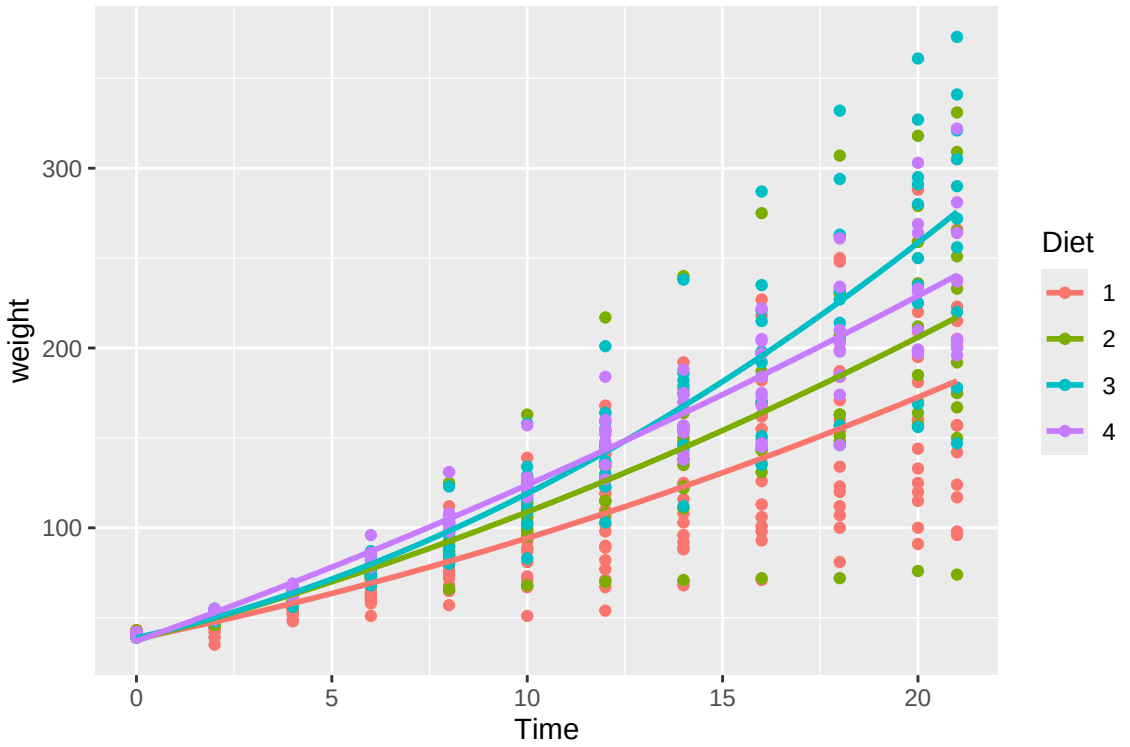


Figure 13.19: Scatterplot of chick weights, coloured by diet, plotted against time with second order linear least squares regression lines for each diet.

13.2 Graphics with the lattice package

Some possibilities of the **lattice** package [19] will be presented in a presentation.

In case you would like to learn more about the **lattice** package, go to the following web address https://dx.doi.org/10.1007/978-0-387-75969-2, while being connected to the Wageningen University Wireless network named Eduroam, and download the book "Lattice: Multivariate Data Visualization with R" as pdf- or epub-file for free.

Lesson 14

Dynamic Reports . . . continued, and Shiny

14.1 Dynamic Reports . . . continued

In Lesson 4 Section 4.1 you have learned about the basics of writing Dynamic Reports in R Markdown. Remind yourself that within RStudio help about R Markdown and Markdown is available in the top menu via **Help** → **Markdown Quick Reference**, and **Help** → **Cheatsheets** → **R Markdown Cheat Sheet/R Markdown Reference Guide**.

In this section you will advance your knowledge about writing Dynamic Reports with R Markdown language. To get to understand the details of Dynamics Reports, consult [8]. You can also write books and technical documents with R Markdown, to learn more about this topic consult: https://bookdown.org/yihui/bookdown/

14.1.1 YAML header

In the YAML ("YAML Ain't Markup Language") header of an R Markdown file you specify details about your document and what is produced when you push the "Knit" button. Here you see an extended example of the YAML header of a R Markdown document entitled `"My First report (extended)"`.

```
---
title: "My First R report (extended)"
author: "Your Name"
date: "January 31, 2018" # or \today to put the date of knitting the document
output:
  pdf_document:
    fig_caption: yes
    highlight: default
    number_sections: yes
    toc: yes
    toc_depth: 3
documentclass: article
classoption: a4paper
fontsize: 10pt
geometry: margin=2cm
---
```

You can see that lots of options have been added:

- `fig_caption: yes`, controls whether figures are rendered with captions. Possible values are false/no or true/yes (this is false by default).
- `highlight: default`, controls the syntax highlighting style. Supported styles include "default", "tango", "pygments", "kate", "monochrome", "espresso", "zenburn", and "haddock" (specify null to prevent syntax highlighting). How the various highlight styles appear after knitting your document can be seen here: https://eranraviv.com/syntax-highlighting-style-in-rmarkdown/.

- `number_sections: yes`, controls whether sections should be numbered. Possible values are false/no or true/yes.
- `toc: yes`, controls whether a table of contents is generated. Possible values are false/no or true/yes.
- `toc_depth: 3`, specifies the depth of headers applied for the table of contents. If the table of contents depth is not explicitly specified then it defaults to 3 (meaning that all level 1, 2, and 3 headers will be included in the table of contents).
- `documentclass: article`, specifies the LaTeX document class. Explanation about LaTeX is beyond the scope of this course, although learning LaTeX is definitely worth the effort!
- `classoption: a4paper`, set the paper size to A4 instead of the default American standard letter.
- `fontsize: 10pt`, specifies the LaTeX font size with 10pt as default. Other possible sizes are 11pt and 12pt.
- `geometry: margin=2cm`, specifies the margins around the paper to use. Here all margins are set to 2cm, but you can specify each margin separately if wanted *e.g.* `geometry: top=0.5cm,bottom=0.5cm, left=2cm,right=1cm`.

Setting the `classoption` and `geometry` with these values, create more space for your output on a sheet of paper than using the default setting.

Many of the settings specified above can also be set by using the "Settings" button next to the "Knit" button in your code editor window of RStudio. Select the option "Output Options...".

For many more options check the following website: https://bookdown.org/yihui/rmarkdown/pdf-document.html

14.1.2 Code chunk-label

As was explained in Lesson 4 Section 4.1 each code chunk, starting with ```` ```{r chunk-label} ````, **must** have a unique chunk-label. It is considered good practice to use the `label` argument in a code chunk to specify the chunk-label, *e.g.* ```` ```{r label="chunk-label"} ````.

For specification of labels (chunk-labels) the same coding style is advised as for object names, as described in Section 6.1.1.1. The reason, why labels (chunk-labels) are important, is that they are used for cross-referencing in your R Markdown document. This will be shown in Section 14.1.7.

14.1.3 Inline R code

In your R Markdown document there is no need to retype values, you have calculated in R. Retyping values is dangerous, a typographical error can easily slip in. The whole idea of R Markdown is to create reproducible research and prevent errors arising from retyping values.

In text you can use inline R code to calculate values or retrieve object values from your workspace environment. For example let's build linear model:

```
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2, 10, 20, labels = c("Ctl","Trt"))
weight <- c(ctl, trt)
lmD9 <- lm(weight ~ group)
```

Now you can retrieve the Analysis of Variance table and store in your workspace environment:

```
aovTable <- anova(lmD9)
```

The Analysis of Variance table is shown in Table 14.1.

Table 14.1: Analysis of Variance of lmD9

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
group	1	0.6882	0.6882	1.4191	0.249

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Residuals	18	8.7292	0.4850	NA	NA

The probability for a quantile from a F-distribution with numerator degrees of freedom = 1, and denominator degrees of freedom = 18, as given in Table 14.1, can be extracted, rounded to 5 digits, with:
`` `r round(aovTable[["Pr(>F)"]][1], digits = 5)` ``
and the value will be printed in your knitted document as: 0.24902.

14.1.4 Equations

It is possible to create nicely formatted formulas/equations in your R Markdown document. There are multiple types you can create:

- inline (within text) formulas/equations
- displayed formulas/equations
- displayed and automatically numbered formulas/equations

The writing of the formulas/equations themselves is done in LaTeX style. A good reference to learn this is found here: https://en.wikibooks.org/wiki/LaTeX/Mathematics. It is not too difficult, but it will take a bit of trial and error to get skillful at it.

> Tip: Use inline R code within equations
>
> Inline R code, as described in Section 14.1.3, can be used inside equations to calculate or retrieve results from calculations. This is what makes R markdown so powerful and prevents you from mistakes caused by retyping values.
>
> For example using Table 14.1:
> `` The $p$-value $\approx `r round(aovTable[["Pr(>F)"]][1], digits = 3)`$. ``
> will be printed in your knitted document as:
> The p-value $\approx$ 0.249.

14.1.4.1 Inline formulas/equations

Inline formulas/equations are displayed within a line of text. To properly display an inline formula/equation you have place your formula/equation on the . . . position within `$...$`.

For example:
`The binomial distribution density is given by $P(y=k)=\binom{n}{k} p^k (1-p)^{n-k}$,`
will be displayed in your knitted document as:
The binomial distribution density is given by $P(y = k) = \binom{n}{k} p^k (1-p)^{n-k}$

14.1.4.2 Displayed formulas/equations

You can also create displayed formulas/equations. Displayed formulas/equations reside on their line and are shown centered on the page. To create a displayed formulas/equation place your formula/equation on the . . . position within `$$...$$`.

For example:
`$$\chi^2 = \sum\limits^k\limits_{i=1}\frac{(n_i - E_i)^2}{E_i}$$,`
will be shown in your knitted document as:

$$\chi^2 = \sum\limits^k\limits_{i=1}\frac{(n_i - E_i)^2}{E_i}$$

14.1.4.3 Displayed and automatically numbered formulas/equations

To create a displayed and automatically numbered formula/equation you have to place your formula/equation in a, so-called, equation environment. The equation environment starts with `\begin{equation}` and ends with `\end{equation}`.

For example the pooled sample variance equation in your R Markdown document looks like:

```
\begin{equation}\label{eqn:pooledVariance}
 s_{p}^2 = \frac{(n_{1} - 1)s_{1}^2 + (n_{2} - 1)s_{2}^2}{n_{1} + n_{2} - 2}
\end{equation}
```

and will be depicted in your knitted document as:

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2} \tag{14.1}$$

You see that the formula/equation is shown in the same style as a displayed formula/equation with the addition of numbering on the far right side. Notice that behind the start of the equation environment a label has been added as `\label{eqn:pooledVariance}`. This will be used later in cross-referencing, as explained in Section 14.1.7.

14.1.4.4 Some useful basic mathematics commands

> **!** Important: Only in Formulas/Equations
>
> The following basic mathematics commands (Table 14.2) can only be used within formulas/equations.

Table 14.2: Mathematical symbols for Formulas/Equations

Command	Result
`A \cup B`	$A \cup B$
`A \cap B`	$A \cap B$
`x \in A`	$x \in A$
`5 \pm 2`	5 ± 2
`3 \times x`	$3 \times x$
`z \sim N(0,1)`	$z \sim N(0,1)$
`x \neq 3`	$x \neq 3$
`x \leq 3`	$x \leq 3$
`x \geq 3`	$x \geq 3$
`P(z \geq 4.541) \approx 0`	$P(z \geq 4.541) \approx 0$
`\log{x}`	$\log x$
`\sin{x}`	$\sin x$
`\sqrt{x}`	$\sqrt{x}$
`\bar{y}`	$\bar{y}$
`\hat{y}`	$\hat{y}$
`\frac{x}{y}`	$\frac{x}{y}$
`\binom{n}{k}`	$\binom{n}{k}$
`x^{2t}`	x^{2t}
`x_{2t}`	x_{2t}
`\displaystyle \sum\limits^{k}\limits_{i=1}`	$\displaystyle\sum\limits^{k}\limits_{i=1}$
`s_{\bar{y}} = \frac{s_{y}}{\sqrt{n}}`	$s_{\bar{y}} = \frac{s_y}{\sqrt{n}}$

In order for some operators, such as `\lim` or `\sum` to be displayed correctly inside inline formulas/equations, it might be convenient to write the `\displaystyle` class inside the environment. This is shown in some examples

above. Doing so might cause the line to be taller, but will cause exponents and indices to be displayed correctly for some math operators.

14.1.4.5 The Greek alphabet

For those of you who don't know the Greek alphabet, it's time to learn it. Greek letters are frequently used in statistical formulas/equations. Also for the commands for the Greek alphabet given below, holds that they only work in a formula/equation.

Command	Result
\alpha	α
\beta	β
\gamma, \Gamma	γ, Γ
\delta, \Delta	δ, Δ
\epsilon, \varepsilon	ϵ, ε
\zeta	ζ
\eta	η
\theta, \vartheta, \Theta	θ, ϑ, Θ
\iota	ι
\kappa	κ
\lambda, \Lambda	λ, Λ
\mu	μ
\nu	ν
\xi, \Xi	ξ, Ξ
\omicron	o
\pi, \Pi	π, Π
\rho, \varrho	ρ, ϱ
\sigma, \varsigma, \Sigma	σ, ς, Σ
\tau	τ
\upsilon, \Upsilon	υ, Υ
\phi, \varphi, \Phi	ϕ, φ, Φ
\chi	χ
\psi, \Psi	ψ, Ψ
\omega, \Omega	ω, Ω

14.1.5 Tables

It is easy to create a table in your R Markdown from objects in your workspace environment. In Section 14.1.3 an Analysis of Variance object named ***aovTable*** was created. This object already has a table like structure:

```
print(aovTable)
```

```
#> Analysis of Variance Table
#>
#> Response: weight
#>           Df Sum Sq Mean Sq F value Pr(>F)
#> group      1 0.6882 0.68820  1.4191  0.249
#> Residuals 18 8.7292 0.48496
```

To make a table out of this object you put the following command **inside a code chunk** in your R Markdown document:

```
knitr::kable(x = aovTable,
             digits = 4,
             caption = "\\label{tab:aovTable}Analyis of Variance of lmD9")
```

After knitting your document a nicely formatted table with a caption will appear as shown in Table 14.1. Notice that the caption contains `\label{tab:aovTable}`. This is the label to be used for cross-referencing the created table, as will be explained in Section 14.1.7.

14.1.6 Figures and images

So far you have been able to include plots made in R into your R Markdown document. Knitting the document will produce a nice figure in your created pdf. However, you have not been able to control the size (width and height) and position of the figure, put a caption, control the placement of the figure on the page.

All these settings are controlled by chunk options. Chunk options are added just like the chunk-label, as described in Section 14.1.2. All chunk options are separated by commas. A description of all possible chunk options can be found here:https://yihui.name/knitr/options/

The most used options for figures and images are:

- `fig.align="center"`, to center a figure on the page. Other possible values are left, right, default.
- `fig.width=value`, where value is a numerical value in inches. This controls the figure width.
- `fig.height=value`, where value is a numerical value in inches. This controls the figure height.
- `fig.pos="!htbp"`, this controls the vertical placement of a figure on a page. For example `"!ht"` means to overwrite all settings (`"!"`), place it here (`"h"`) or float the figure to the top of the page (`"t"`). The `"b"` means float to the bottom, and the `"p"` means float to a separate page. Normally it always starts with `"!h"` followed with another vertical position letter.
- `fig.cap="Put your figure caption here"`, lets you specify the figure caption to be put under your figure.
- `fig.scap="Put a short caption here"`, if your figure caption (in `fig.cap`) is very long, you can specify a short caption here. This short caption will be used in the list of figures, you can create with `\listoffigures` somewhere at the end of your document. You can leave this option empty by putting the following as chunk option: `fig.scap=NA`

Instead of creating figures with plotting commands, such as *plot()*, you can also include images. For this you can use the following command inside a code chunk:

```
include_graphics("path to your image")
```

For the figure caption and alignment you can use the options as described above for figures. However, the easiest chunk option to control the size of the image, with keeping the aspect ratio, is the `out.width="value"` or `out.height="value"` chunk option. The value can be specified as a percentage, *e.g.* '50%'. The following code produced Figure 14.1:

````
```{r label="slothImage", echo=FALSE, eval=TRUE, fig.cap="A beautifull sloth image",
fig.scap=NA, fig.align='center', out.width='50%'}
knitr::include_graphics("figures/sloth1.png")
```
````

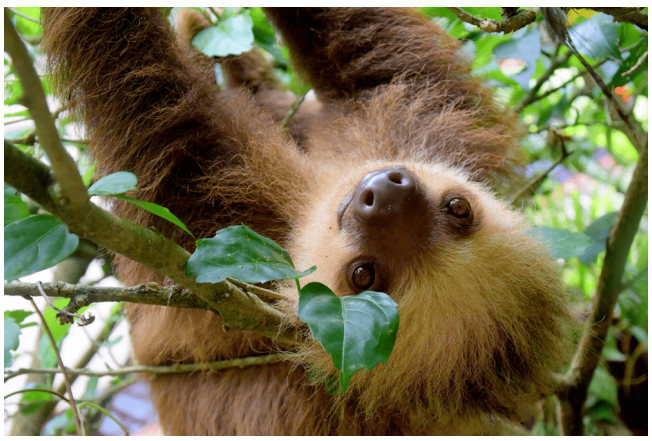


Figure 14.1: A beautifull sloth image.

14.1.7 Cross-references

14.1.7.1 Sections

To cross-reference a section in your R Markdown document you first need to define a label. In R Markdown for sections this can be done by placing `{#labelid}` behind your section header. For example for a chapter heading it could look like this: `# Assignment Week 1 {#chp:assignWeekOne}`. For other sections, e.g. a subsection about simulation in a results section, it could look like this:
`### Simulation {#sec:simulationResults}`.

To create a cross-reference to a section in your document place a reference in your text, like this:
`\ref{labelid}`. For example to cross-reference to the chapter and subsection, previously given as example, the textline in your R Markdown document would look like this:
`Chapter \ref{chp:assignWeekOne} Section \ref{sec:simulationResults} clearly shows that...`

14.1.7.2 Displayed and numbered formulas/equation

To cross-reference a displayed and numbered formula/equation, as created in Section 14.1.4, place a reference like `\ref{equationLabel}` in your R Markdown document. Here the equation label must match the one specified with `\label{}` after `\begin{equation}`. As an example to cross reference the pooled variance equation you would add a sentence like this in your R Markdown document:
`The pooled sample variance equation shown by Equation \ref{eqn:pooledVariance}.`
will appear after knitting the document as:
The pooled sample variance equation shown by Equation 14.1.

14.1.7.3 Tables

Tables can be cross-referenced to in the same way as equations. Place a `ref{}` to the table labelid, as specified by `\label{}` in the `caption` argument of the *knitr::kable()* function.

For example to cross-reference the ANOVA table, shown in Table 14.1 and created in Section 14.1.5 , you would have to use in your text:
`See Table \ref{tab:aovTable} for the ANOVA table.`

14.1.7.4 Figures and images

In order to cross-reference a figure or image you also have to use `\ref{labelid}`. However, the figure/image labelid is dependent on the chunk option `fig.lp` of the chunk where the figure or image was defined. By default the figure label prefix, hopefully explaining the name of the chunk option, is `fig:`. Therefore, in R Markdown the cross-reference always, unless you have redefined the option `fig.lp`, will take the following form: `\ref{fig:chunk-label}`.

For example a cross-reference to the beautiful sloth image in your R Markdown could look like:
`Sloths, as depicted in Figure \ref{fig:slothImage}, are immensely cute animals.`
after rendering (a.k.a. knitting) the document it would appear like:
Sloths, as depicted in Figure 14.1, are immensely cute animals.

14.2 Shiny

Shiny [26] is an R package that makes it easy to build interactive web applications (apps) straight from R. This lesson will get you started building Shiny apps right away. The contents of this lesson have been taken from the internet, and can be found at https://shiny.posit.co/r/getstarted/shiny-basics/lesson1/ under **Shiny for R** → **Get Started**. Only the first seven basic lessons of these written tutorials have been included, the rest you can read online.

If you still haven't installed the Shiny package, open an R session, connect to the internet, and run:

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

14.2.1 Examples

The **shiny** package has eleven built-in examples that each demonstrate how Shiny works. Each example is a self-contained Shiny app.

You can create Shiny apps by copying and modifying existing Shiny apps. The Shiny gallery (https://shiny.rstudio.com/gallery/) provides some good examples, or use the eleven built-in Shiny examples listed below.

```
runExample("01_hello")      # a histogram
runExample("02_text")       # tables and data frames
runExample("03_reactivity") # a reactive expression
runExample("04_mpg")        # global variables
runExample("05_sliders")    # slider bars
runExample("06_tabsets")    # tabbed panels
runExample("07_widgets")    # help text and submit buttons
runExample("08_html")       # Shiny app built from HTML
runExample("09_upload")     # file upload wizard
runExample("10_download")   # file download wizard
runExample("11_timer")      # an automated timer
```

Each demonstrates a feature of Shiny apps. All Shiny example apps open in `"showcase"` mode with the `app.R` script in the display, as will be explained in Section 14.2.4.

But why limit yourself to copying other apps? The following sections will show you how to build your own Shiny apps from scratch. You'll learn about each part of a Shiny app, and finish by deploying your own Shiny app online.

Figure 14.2: Example of a shiny app.

The **Hello Shiny** example, shown in Figure 14.2, plots a histogram of R's ***faithful*** data set object with a configurable number of bins. Users can change the number of bins with a slider bar, and the app will immediately respond to their input. You'll use **Hello Shiny** to explore the structure of a Shiny app and to create your first app.

To run **Hello Shiny**, type:

```
library(shiny)
runExample("01_hello")
```

14.2.2 Structure of a Shiny App

Shiny apps are contained in a single script called `app.R`. The script `app.R` lives in a directory (for example, `newdir/`) and the app can be run with `runApp("newdir")`.

`app.R` has three components:

- a user interface object
- a server function

- a call to the shinyApp function

The user interface (***ui***) object controls the layout and appearance of your app. The *server()* function contains the instructions that your computer needs to build your app. Finally the *shinyApp()* function creates Shiny app objects from an explicit UI/server pair.

> **i** Note: Shiny single-file apps
>
> Prior to version 0.10.2, Shiny did not support single-file apps and the ***ui*** object and *server()* function needed to be contained in separate scripts called ui.R and server.R, respectively. This functionality is still supported in Shiny, however the tutorial and much of the supporting documentation focus on single-file apps.
>
> One nice feature about single-file apps is that you can copy and paste the entire app into the R console, which makes it easy to quickly share code for others to experiment with. For example, if you copy and paste the code above into the R command line, it will start a Shiny app.

14.2.2.1 ui (user interface)

Here is the ***ui*** object for the **Hello Shiny** example.

```
library(shiny)
# Define UI for app that draws a histogram ----
ui <- fluidPage(
  # App title ----
  titlePanel("Hello Shiny!"),

  # Sidebar layout with input and output definitions ----
  sidebarLayout(
    # Sidebar panel for inputs ----
    sidebarPanel(
      # Input: Slider for the number of bins ----
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),

    # Main panel for displaying outputs ----
    mainPanel(
      # Output: Histogram ----
      plotOutput(outputId = "distPlot")
    )
  )
)
```

14.2.2.2 server

Here is the *server()* function for the **Hello Shiny** example.

```
# Define server logic required to draw a histogram ----
server <- function(input, output) {
  # Histogram of the Old Faithful Geyser Data ----
  # with requested number of bins
  # This expression that generates a histogram is wrapped in a call
  # to renderPlot to indicate that:
  #
  # 1. It is "reactive" and therefore should be automatically
```

```
  #    re-executed when inputs (input$bins) change
  # 2. Its output type is a plot
  output$distPlot <- renderPlot({
    x    <- faithful$waiting
    bins <- seq(min(x), max(x), length.out = input$bins + 1)
    hist(x,
         breaks = bins,
         col = "#75AADB",
         border = "white",
         xlab = "Waiting time to next eruption (in mins)",
         main = "Histogram of waiting times")
    })
}
```

At one level, the **Hello Shiny** *server()* function is very simple. The script does some calculations and then plots a histogram with the requested number of bins.

However, you'll also notice that most of the script is wrapped in a call to *renderPlot()*. The comment above the function explains a bit about this, but if you find it confusing, don't worry. This concept will be covered in much more detail soon.

Play with the **Hello Shiny** app and review the source code. Try to develop a feel for how the app works. But before you do so, note that in your `app.R` file you will need to start with loading the Shiny package and end with a call to `shinyApp`:

```
library(shiny)
# See above for the definitions of ui and server
ui <- ...
server <- ...
shinyApp(ui = ui, server = server)
```

Your R session will be busy while the **Hello Shiny** app is active, so you will not be able to run any R commands. R is monitoring the app and executing the app's reactions. To get your R session back, hit escape or click the stop sign icon (found in the upper right corner of the RStudio console panel).

14.2.3 Running an App

Every Shiny app has the same structure: an `app.R` file that contains ***ui*** and *server()*. You can create a Shiny app by making a new directory and saving an `app.R` file inside it. It is recommended that each app will live in its own unique directory.

You can run a Shiny app by giving the name of its directory to the *runApp()* functionp. For example if your Shiny app is in a directory called `my_app`, run it with the following code:

```
library(shiny)
runApp("my_app")
```

> **i** Note: *runApp()* function
>
> *runApp()* is similar to *read.csv()*, *read.table()*, and many other functions in R. The first argument of *runApp()* is the file path from your working directory to the app's directory. The code above assumes that the app directory is in your working directory. In this case, the file path is just the name of the directory.
>
> (In case you are wondering, the **Hello Shiny** app's files are saved in a special system directory called "`01_hello`". This directory is designed to work with the `runExample("01_hello")` call.)

14.2.4 Exercise Shiny: Modifying the Hello Shiny App

Create a new directory named `App-1` in your working directory. Then copy and paste the `app.R` script above into your directory (the scripts from **Hello Shiny**). When you are finished the directory should look like this:

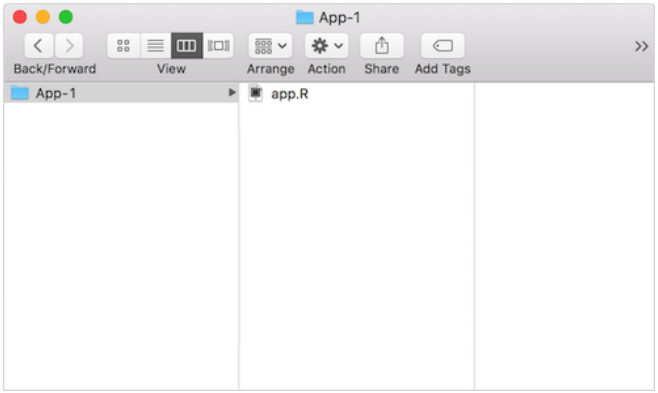


Figure 14.3: Directory structure for modifying the **Hello Shiny** app.

Launch your app by running `runApp("App-1")`. Then click escape and make some changes to your app:

1. Change the title from "Hello Shiny!" to "Hello World!".
2. Set the minimum value of the slider bar to 5.
3. Change the histogram border color from "white" to "orange".

When you are ready, launch your app again. Your new app should match the image below (Figure 14.4).

Figure 14.4: Example of the modified Hello Shiny app.

By default, Shiny apps display in "normal" mode, like the app pictured above. **Hello Shiny** and the other built in examples display in "showcase mode", a different mode that displays the `app.R` script alongside the app.

If you would like your app to display in showcase mode, you can run `runApp("App-1", display.mode = "showcase")`.

14.2.5 Relaunching Apps

To relaunch your Shiny app:

- `runApp("App-1")`, or
- Open the `app.R` script in your RStudio editor. RStudio will recognize the Shiny script and provide a Run App button (at the top of the editor as shown in Figure 14.5). Either click this button to launch your app or use the keyboard shortcut: Control+Shift+Enter (on Windows and Linux) Command+Shift+Enter (on Mac).

RStudio will launch the app in a new window by default, but you can also choose to have the app launch in a dedicated viewer pane, or in your external web browser. Make your selection by clicking the icon next to Run App (Figure 14.6).

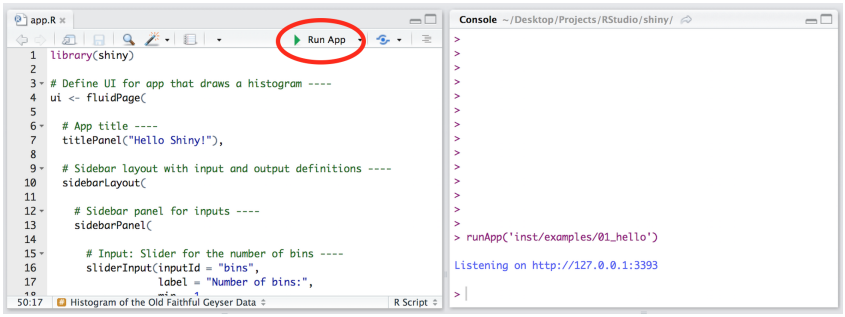


Figure 14.5: Run an app from RStudio.

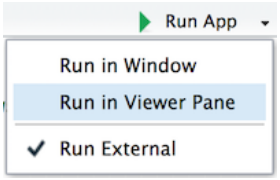


Figure 14.6: Launch options to run an app from RStudio.

14.2.6 Build a user interface (ui)

Now that you understand the structure of a Shiny app, it's time to build your first app from scratch.

This section will show you how to build a user interface for your app. You will learn how to lay out the user interface and then add text, images, and other HTML elements to your Shiny app.

The `App-1` app you made in the previous sections will be used. To get started, open its `app.R` file. Edit the script to match the one below:

```
library(shiny)
# Define UI ----
ui <- fluidPage(
  ...
)

# Define server logic ----
server <- function(input, output) {
  ...
}

# Run the app ----
shinyApp(ui = ui, server = server)
```

This code is the bare minimum needed to create a Shiny app. The result is an empty app with a blank user interface, an appropriate starting point for this section.

14.2.6.1 Layout

Shiny uses the *fluidPage()* function to create a display that automatically adjusts to the dimensions of your user's browser window. You lay out the user interface of your app by placing elements in the *fluidPage()* function.

For example, the ***ui*** object below creates a user interface that has a title panel and a sidebar layout, which includes a sidebar panel and a main panel (displayed in Figure 14.7). Note that these elements are placed within the *fluidPage()* function.

```
ui <- fluidPage(
  titlePanel("title panel"),
  sidebarLayout(sidebarPanel("sidebar panel"),
                mainPanel("main panel")
```

```
  )
)
```

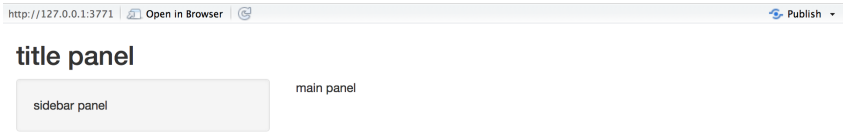


Figure 14.7: User Interface containing the following panels: title, sidebar and main.

titlePanel() and *sidebarLayout()* are the two most popular elements to add to *fluidPage()*. They create a basic Shiny app with a sidebar.

sidebarLayout() always takes two arguments:

- *sidebarPanel()* function output
- *mainPanel()* function output

These functions place content in either the sidebar or the main panels.

The sidebar panel will appear on the left side of your app by default. You can move it to the right side by giving *sidebarLayout()* the optional argument `position = "right"`:

```
ui <- fluidPage(
  titlePanel("title panel"),
  sidebarLayout(position = "right",
                sidebarPanel("sidebar panel"),
                mainPanel("main panel")
  )
)
```

titlePanel() and *sidebarLayout()* create a basic layout for your Shiny app, but you can also create more advanced layouts. You can use *navbarPage()* to give your app a multi-page user interface that includes a navigation bar. Or you can use *fluidRow()* and *column()* to build your layout up from a grid system. If you'd like to learn more about these advanced options, read the Shiny Application Layout Guide (https://shiny.rstudio.com/articles/layout-guide.html). In these sections the *sidebarLayout()* will be used as default.

14.2.6.2 HTML Content

You can add content to your Shiny app by placing it inside a **Panel()* function. For example, the apps above display a character string in each of their panels. The words "sidebar panel" appear in the sidebar panel, because the string was added to the *sidebarPanel()* function, *e.g.* `sidebarPanel("sidebar panel")`. The same is true for the text in the title panel and the main panel.

To add more advanced content, use one of Shiny's HTML tag functions. These functions parallel common HTML5 tags. Let's try out a few of them.

shiny function	HTML5 equivalent	creates
p()	`<p>`	A paragraph of text
h1()	`<h1>`	A first level header
h2()	`<h2>`	A second level header

shiny function	HTML5 equivalent	creates
h3()	`<h3>`	A third level header
h4()	`<h4>`	A fourth level header
h5()	`<h5>`	A fifth level header
h6()	`<h6>`	A sixth level header
a()	`<a>`	A hyper link
br()	` `	A line break
div()	`<div>`	A division of text with uniform style
span()	`<span>`	An in-line division of text with a uniform style
pre()	`<pre>`	Text 'as is' in a fixed width font
code()	`<code>`	A formatted block of code
img()	`<img>`	An image
strong()	`<strong>`	Bold text
em()	`<em>`	Italicized text
HTML()		Directly passes a character string as HTML code

14.2.6.3 Headers

To create a header element:

- select a header function (e.g., h1 or h5),
- give it the text you want to see in the header.

For example, you can create a first level header that says "My title" with `h1("My title")`. If you run the command at the command line, you'll notice that it produces HTML code.

```
shiny::h1("My title")
```

To place the element in your app:

- pass `h1("My title")` as an argument to *titlePanel()*, *sidebarPanel()*, or *mainPanel()*

The text will appear in the corresponding panel of your web page. You can place multiple elements in the same panel if you separate them with a comma.

Give this a try. The new script below uses all six levels of headers. Update your `ui.R` to match the script and then relaunch your app. Remember to relaunch a Shiny app you may run `runApp("App-1")`, click the Run App button, or use your keyboard shortcuts.

```
ui <- fluidPage(
    titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      h1("First level title"),
      h2("Second level title"),
      h3("Third level title"),
      h4("Fourth level title"),
      h5("Fifth level title"),
      h6("Sixth level title")
    )
  )
)
```

Now your app should look like shown in Figure 14.8.

If George Lucas had a first app, it might look like shown in Figure 14.9. You can create this effect with `align = "center"`, as in `h6("Episode IV", align = "center")`. In general, any HTML tag attribute can be set as an argument in any Shiny tag function.

If you are unfamiliar with HTML tag attributes, you can look them up in one of the many free online HTML resources such as w3schools.

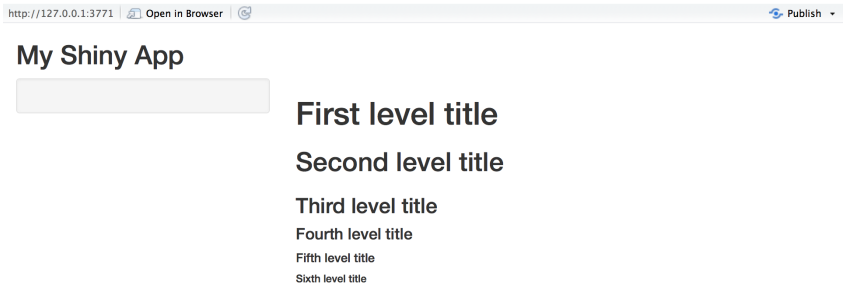


Figure 14.8: Example of using headers in the mainpanel in a shiny app.

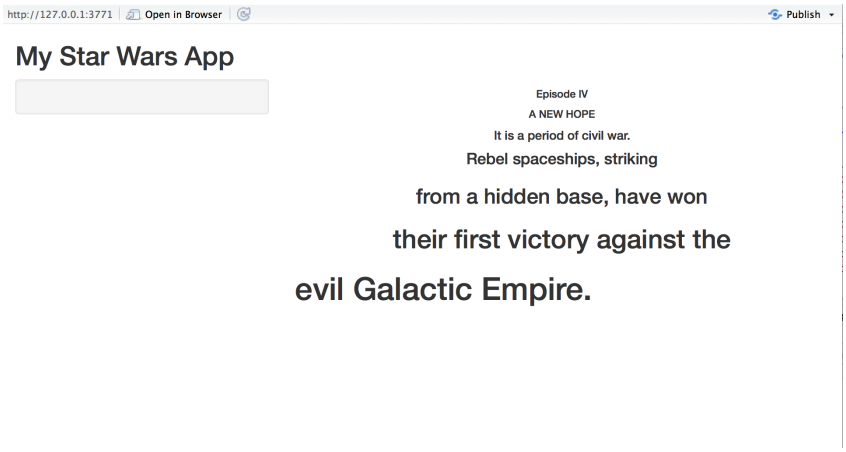


Figure 14.9: Use of headers in the Star Wars App.

Here's the code for the ***ui*** that made the Star Wars-inspired user interface:

```
ui <- fluidPage(
  titlePanel("My Star Wars App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      h6("Episode IV", align = "center"),
      h6("A NEW HOPE", align = "center"),
      h5("It is a period of civil war.", align = "center"),
      h4("Rebel spaceships, striking", align = "center"),
      h3("from a hidden base, have won", align = "center"),
      h2("their first victory against the", align = "center"),
      h1("evil Galactic Empire.")
    )
  )
)
```

14.2.6.4 Formatted text

Shiny offers many tag functions for formatting text. The easiest way to describe them is by running through an example.

Paste the ***ui*** object below into your `app.R` file and save it. If your Shiny app is still running, you can refresh your web page or preview window, and it will display the changes. If your app is closed, just relaunch it.

Compare the displayed app (Figure 14.10) to your updated ***ui*** object definition to discover how to format text in a Shiny app.

```
ui <- fluidPage(
  titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      p("p creates a paragraph of text."),
      p("A new p() command starts a new paragraph. Supply a style attribute to change
         the format of the entire paragraph.",
        style = "font-family: 'times'; font-si16pt"),
      strong("strong() makes bold text."),
      em("em() creates italicized (i.e, emphasized) text."),
      br(),
      code("code displays your text similar to computer code"),
      div("div creates segments of text with a similar style. This division of text is all
           blue because I passed the argument 'style = color:blue' to div",
          style = "color:blue"),
      br(),
      p("span does the same thing as div, but it works with",
        span("groups of words", style = "color:blue"),
        "that appear inside a paragraph.")
    )
  )
)
```

14.2.6.5 Images

Images can enhance the appearance of your app and help your users understand the content. Shiny looks for the *img()* function to place image files in your app.

To insert an image, give the *img()* function the name of your image file as the `src` argument (e.g., `img(src = "my_image.png")`). You must spell out this argument since *img()* passes your input to an HTML tag, and `src` is what the tag expects.

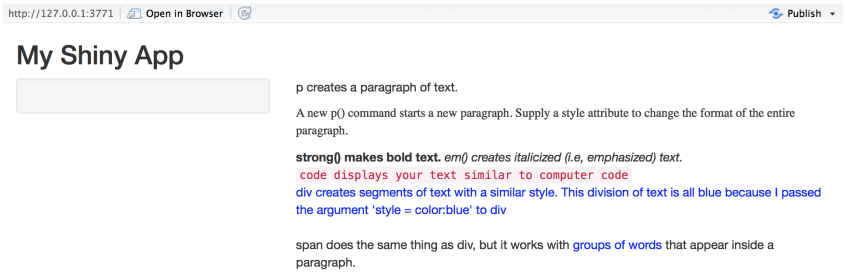


Figure 14.10: Use of formatting in a shiny App.

You can also include other HTML friendly parameters such as height and width. Note that height and width numbers will refer to pixels. For example:

```
img(src = "my_image.png", height = 72, width = 72)
```

The *img()* function looks for your image file in a specific place. Your file **must** be in a folder named `www` in the same directory as the `app.R` script. Shiny treats this directory in a special way. Shiny will share any file placed here with your user's web browser, which makes `www` a great place to put images, style sheets, and other things the browser will need to build the web components of your Shiny app.

So if you want to use an image named `rstudio.png` (download from https://shiny.rstudio.com/tutorial/written-tutorial/lesson2/www/rstudio.png), your `App-1` directory should look like the one shown in Figure 14.11.

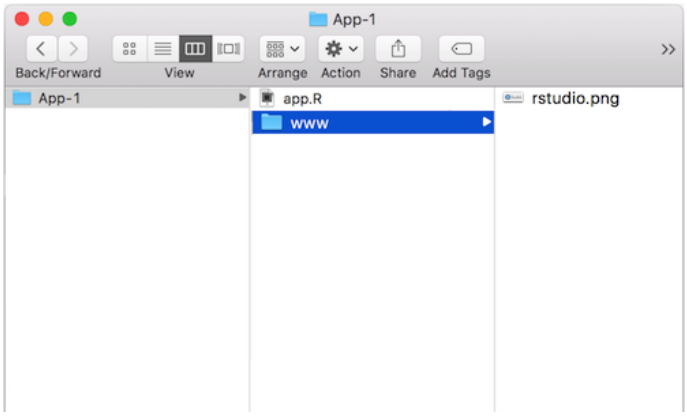


Figure 14.11: Shiny app folder including www folder for images.

With this file arrangment, the ***ui*** object below can create the app displayed in Figure 14.12.

```
ui <- fluidPage(
  titlePanel("My Shiny App"),
  sidebarLayout(
    sidebarPanel(),
    mainPanel(
      img(src = "rstudio.png", height = 140, width = 400)
    )
  )
)
```

14.2.6.6 Other Tags

So far the most popular Shiny tag functions have been covered, but there are many more tag functions for you to use. You can learn about additional tag functions in "Customize your UI with HTML" (https://shiny.rstudi

Figure 14.12: App with image in mainPanel.

o.com/articles/html-tags.html) and the "Shiny HTML Tags Glossary" (https://shiny.rstudio.com/articles/tag-glossary.html).

14.2.7 Exercise Shiny: Formatting Text in My Shiny App

You can use Shiny's layout, HTML, and *img()* functions to create very attractive and useful user interfaces. See how well you understand these functions by recreating the Shiny app pictured in Figure 14.13.

Use the examples shown so far to work on it and then test it out.

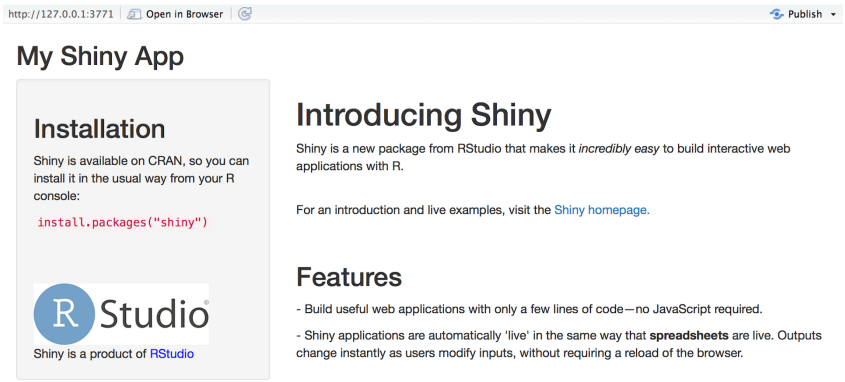


Figure 14.13: My Shiny app to practice formatting.

14.2.8 Add control widgets

This section will show you how to add control widgets to your Shiny apps. What's a widget? A web element that your users can interact with. Widgets provide a way for your users to send messages to the Shiny app.

Shiny widgets collect a value from your user. When a user changes the widget, the value will change as well. This sets up opportunities for displaying reactive output in your Shiny app.

14.2.8.1 Control widgets

Shiny comes with a family of pre-built widgets, each created with a transparently named R function. For example, Shiny provides a function named *actionButton()* that creates an Action Button and a function named *sliderInput()* that creates a slider bar.

The standard Shiny widgets are:

function	widget
actionButton()	Action Button
checkboxGroupInput()	A group of check boxes
checkboxInput()	A single check box
dateInput()	A calendar to aid date selection
dateRangeInput()	A pair of calendars for selecting a date range
fileInput()	A file upload control wizard
helpText()	Help text that can be added to an input form
numericInput()	A field to enter numbers
radioButtons()	A set of radio buttons
selectInput()	A box with choices to select from
sliderInput()	A slider bar
submitButton()	A submit button
textInput()	A field to enter text

Figure 14.14 shows how the widgets look like in the default Shiny user interface.

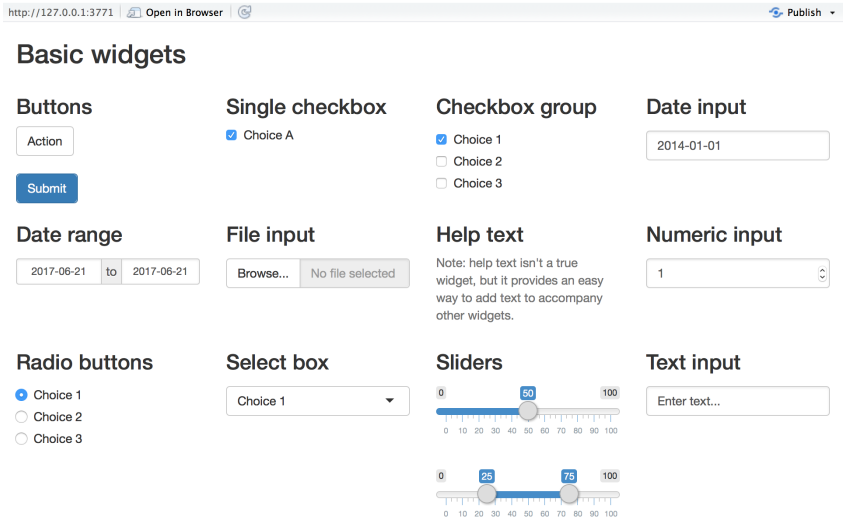


Figure 14.14: Visualization of all basic widgets in an app.

14.2.8.2 Adding widgets

You can add widgets to your web page in the same way that you added other types of HTML content as described in Section 14.2.6.2. To add a widget to your app, place a widget function in *sidebarPanel()* or *mainPanel()* in your ***ui*** object.

Each widget function requires several arguments. The first two arguments for each widget are

- a **name for the widget**: The user will not see this name, but you can use it to access the widget's value. The name should be a character string.
- a **label**: This label will appear with the widget in your app. It should be a character string, but it can be an empty string `""`.

In this example, the name is "action" and the label is "Action": `actionButton("action", label = "Action")`.

The remaining arguments vary from widget to widget, depending on what the widget needs to do its job. They include things like initial values, ranges, and increments. You can find the exact arguments needed by a widget on the widget function's help page, (*e.g.*, `?selectInput`).

The `app.R` script below makes the app pictured in Figure 14.14. Change your own `App-1/app.R` script to match it, and then launch the app (`runApp("App-1")`, select Run App, or use shortcuts).

Play with each widget to get a feel for what it does. Experiment with changing the values of the widget functions and observe the effects. If you are interested in the layout scheme for this Shiny app, read the description in the "application layout guide" (https://shiny.rstudio.com/articles/layout-guide.html). This section about Shiny will not cover this slightly more complicated layout scheme, but it is interesting to note what it does.

```
library(shiny)
# Define UI ----
ui <- fluidPage(
  titlePanel("Basic widgets"),
  fluidRow(
    column(3,
           h3("Buttons"),
           actionButton("action", "Action"),
           br(),
           br(),
           submitButton("Submit")),
    column(3,
           h3("Single checkbox"),
           checkboxInput("checkbox", "Choice A", value = TRUE)),
    column(3,
           checkboxGroupInput("checkGroup",
                              h3("Checkbox group"),
                              choices = list("Choice 1" = 1,
                                             "Choice 2" = 2,
                                             "Choice 3" = 3),
                              selected = 1)),
    column(3,
           dateInput("date",
                     h3("Date input"),
                     value = "2014-01-01"))
  ),
  fluidRow(
    column(3,
           dateRangeInput("dates", h3("Date range"))),
    column(3,
           fileInput("file", h3("File input"))),
    column(3,
           h3("Help text"),
           helpText("Note: help text isn't a true widget,",
                    "but it provides an easy way to add text to",
                    "accompany other widgets.")),
    column(3,
           numericInput("num",
                        h3("Numeric input"),
                        value = 1))
  ),
  fluidRow(
    column(3,
           radioButtons("radio", h3("Radio buttons"),
                        choices = list("Choice 1" = 1, "Choice 2" = 2,
                                       "Choice 3" = 3),selected = 1)),
    column(3,
           selectInput("select", h3("Select box"),
                       choices = list("Choice 1" = 1, "Choice 2" = 2,
                                      "Choice 3" = 3), selected = 1)),
    column(3,
           sliderInput("slider1", h3("Sliders"),
```

```
                      min = 0, max = 100, value = 50),
          sliderInput("slider2", "",
                      min = 0, max = 100, value = c(25, 75))
    ),
    column(3,
           textInput("text", h3("Text input"),
                     value = "Enter text..."))
  )
)
# Define server logic ----
server <- function(input, output) {
}
# Run the app ----
shinyApp(ui = ui, server = server)
```

14.2.9 Exercise Shiny: Adding Control Widgets

Rewrite your ***ui*** object to create the user interface depicted in Figure 14.15 and Figure 14.16. Notice that this Shiny app uses a basic Shiny layout (no columns) and contains three of the widgets pictured above. The other values of the select box are shown below the image of the app.

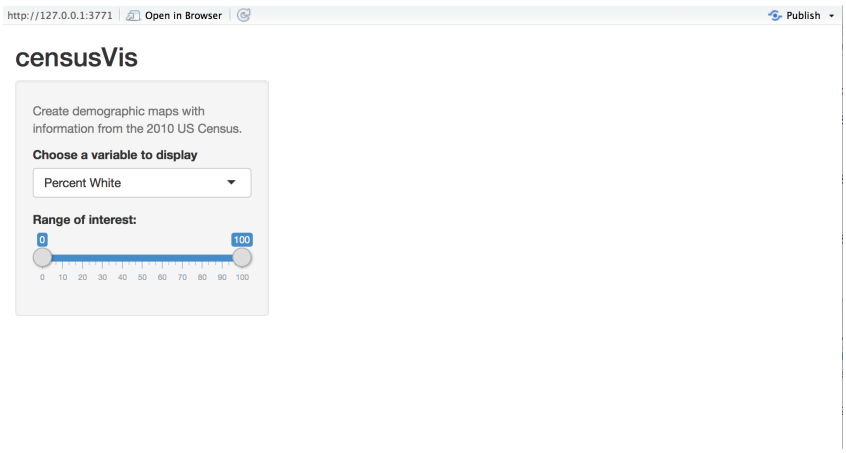


Figure 14.15: UI to recreate in Exercise Shiny: Adding Control Widgets.

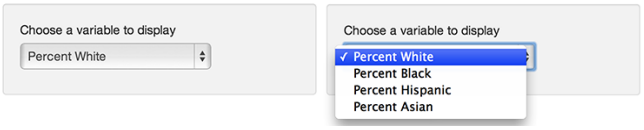


Figure 14.16: Selection box options.

14.2.10 Display reactive output

Time to give your Shiny app a "live" quality! This section will teach you how to build reactive output to displays in your Shiny app. Reactive output automatically responds when your user toggles a widget.

By the end of this section, you'll know how to make a simple Shiny app with two reactive lines of text as depicted in Figure 14.17. Each line will display the values of a widget based on your user's input.

This new Shiny app will need its own, new directory. Create a folder in your working directory named `census-app`. This is where you will save the `app.R` file that you make in this section.

14.2.10.1 Two steps

You can create reactive output with a two step process:

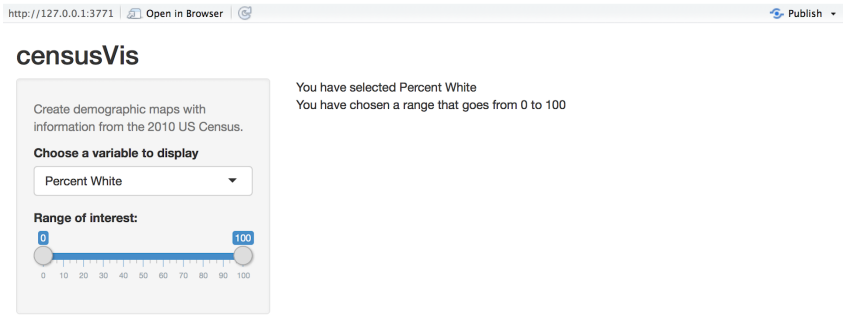


Figure 14.17: User Interface of the censusVis app with two responsive lines in the app's main panel.

- Add an R object to your user interface.
- Tell Shiny how to build the object in the server function. The object will be reactive if the code that builds it calls a widget value.

Step 1: Add an R object to the UI Shiny provides a family of functions that turn R objects into output for your user interface. Each function creates a specific type of output.

Output function	Creates
dataTableOutput()	Data table
htmlOutput()	Raw HTML
imageOutput()	Image
plotOutput()	Plot
tableOutput()	Table
textOutput()	Text
uiOutput()	Raw HTML
verbatimTextOutput()	Text

You can add output to the user interface in the same way that you added HTML elements and widgets. Place the output function inside *sidebarPanel()* or *mainPanel()* in the ***ui*** object.

For example, the ***ui*** object below uses *textOutput()* to add a reactive line of text to the main panel of the Shiny app pictured in Figure 14.17.

```
ui <- fluidPage(
  titlePanel("censusVis"),
  sidebarLayout(
    sidebarPanel(
      helpText("Create demographic maps with
               information from the 2010 US Census."),
      selectInput("var",
                  label = "Choose a variable to display",
                  choices = c("Percent White",
                              "Percent Black",
                              "Percent Hispanic",
                              "Percent Asian"),
                  selected = "Percent White"),
      sliderInput("range",
                  label = "Range of interest:",
                  min = 0, max = 100, value = c(0, 100))
    ),
    mainPanel(
```

```
      textOutput("selected_var")
    )
  )
)
```

Notice that *textOutput()* takes an argument, the character string `"selected_var"`. Each of the **Output()* functions require a single argument: a character string that Shiny will use as the name of your reactive element. Your users will not see this name, but you will use it later.

Step 2: Provide R code to build the object. Placing a function in the ***ui*** object tells Shiny where to display your object. Next, you need to tell Shiny how to build the object.

You do this by providing the R code that builds the object in the *server()* function.

The *server()* function plays a special role in the Shiny process; it builds a list-like object named ***output*** that contains all of the code needed to update the R objects in your app. Each R object needs to have its own entry in the list.

You can create an entry by defining a new element for ***output*** within the *server()* function, like shown below. The element name should match the name of the reactive element that you created in the ***ui***.

In the *server()* function below, `output$selected_var` matches `textOutput("selected_var")` in your ***ui*** object.

```
server <- function(input, output) {
  output$selected_var <- renderText({
    "You have selected this"
  })
}
```

You do not need to explicitly state in the *server()* function to return ***output*** in its last line of code. R will automatically update ***output*** through reference class semantics.

Each entry to ***output*** should contain the output of one of Shiny's *render*()* functions. These functions capture an R expression and do some light pre-processing on the expression. Use the *render*()* function that corresponds to the type of reactive object you are making.

render function	Creates
renderDataTable()	Data table
renderImage()	Images (saved as a link to a source file)
renderPlot()	Plots
renderPrint()	Any printed output
renderTable()	Data frame, matrix, other table like structures
renderText()	Character strings
renderUI()	a Shiny tag object or HTML

Each *render*()* function takes a single argument: an R expression surrounded by braces, `{}`. The expression can be one simple line of text, or it can involve many lines of code, as if it were a complicated function call.

Think of this R expression as a set of instructions that you give Shiny to store for later. Shiny will run the instructions when you first launch your app, and then Shiny will re-run the instructions every time it needs to update your object.

For this to work, your expression should return the object you have in mind (a piece of text, a plot, a data frame, *etc.*). You will get an error if the expression does not return an object, or if it returns the wrong type of object

14.2.10.2 Use widget values

If you run the app with the *server()* function above, it will display "You have selected this" in the main panel. However, the text will not be reactive. It will not change even if you manipulate the widgets of your app.

You can make the text reactive by asking Shiny to call a widget value when it builds the text. Let's look at how to do this.

Take a look at the first line of code in the *server()* function. Do you notice that the *server()* function mentions two arguments, ***input*** and ***output***? You already saw that ***output*** is a list-like object that stores instructions for building the R objects in your app.

input is a second list-like object. It stores the current values of all of the widgets in your app. These values will be saved under the names that you gave the widgets in your ***ui*** object.

So for example, your app has two widgets, one named `"var"` and one named `"range"` (you gave the widgets these names in Section 14.2.10.1). The values of `"var"` and `"range"` will be saved in ***input*** as `input$var` and `input$range`. Since the slider widget has two values (a min and a max), `input$range` will contain a vector of length two giving the start and end value of the range, respectively.

Shiny will automatically make an object reactive if the object uses an ***input*** value. For example, the *server()* function below creates a reactive line of text by calling the value of the select box widget to build the text.

```
server <- function(input, output) {
  output$selected_var <- renderText({
    paste("You have selected", input$var)
  })
}
```

Shiny tracks which outputs depend on which widgets. When a user changes a widget, Shiny will rebuild all the outputs that depend on the widget, using the new value of the widget as it goes. As a result, the rebuilt objects will be completely up-to-date.

This is how you create reactivity with Shiny, by connecting the values of ***input*** to the objects in ***output***. Shiny takes care of all the other details.

14.2.10.3 Launch your app and see the reactive output

When you are ready, update your *server()* function and ***ui*** object to match those above. Then launch your Shiny app by running `runApp("census-app", display.mode = "showcase")` at the command line. Your app should look like the app below in Figure 14.18, and your statement should update instantly as you change the select box widget.

Figure 14.18: User Interface of the censusVis app with two responsive lines in the app's main panel in showcase display mode.

Watch the *server()* portion of the script. When Shiny rebuilds an output, it highlights the code it is running. This temporary highlighting can help you see how Shiny generates reactive output.

14.2.11 Exercise Shiny: Display reactive output

Add a second line of reactive text to the main panel of your Shiny app. This line should display "You have chosen a range that goes from *something* to *something*", and each *something* should show the current minimum (min) or maximum (max) value of the slider widget.

Don't forget to update both your ***ui*** object and your *server()* function.

14.2.12 Use R scripts and data

This section will show you how to load data, R Scripts, and packages to use in your Shiny apps. Along the way, you will build a sophisticated app that visualizes US Census data as displayed in Figure 14.19.

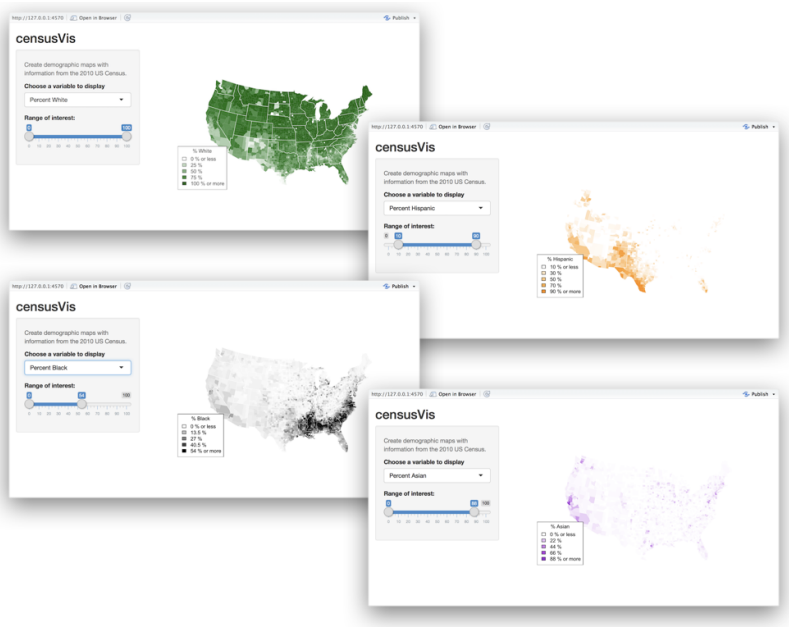


Figure 14.19: Four views on the US census data app.

14.2.12.1 counties.rds

`counties.rds` is a data set of demographic data for each county in the United States, collected with the **UScensus2010** R package. You can download it from the following URL: https://shiny.rstudio.com/tutorial/written-tutorial/lesson5/census-app/data/counties.rds.

Once you have the file,

- Create a new folder named data in your `census-app` directory.
- Move `counties.rds` into the data folder.

When you're done, your `census-app` folder should look like shown in Figure 14.20.

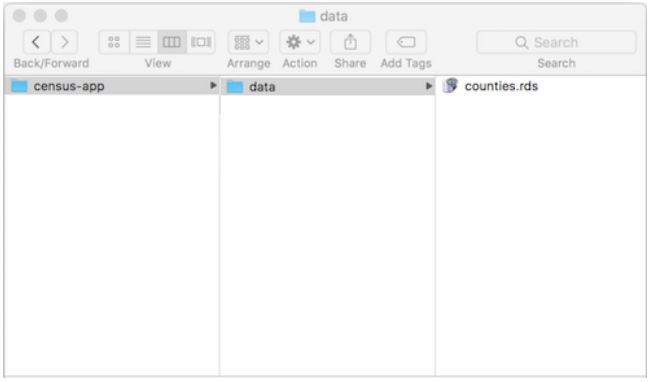


Figure 14.20: Storing data for the US Census data app.

The dataset in `counties.rds` contains:

- the name of each county in the United States,
- the total population of the county,
- the percentage of residents in the county who are White, Black, Hispanic, or Asian

```
counties <- readRDS("census-app/data/counties.rds")
head(counties)
```

```
#>              name total.pop white black hispanic asian
#> 1 alabama,autauga     54571  77.2  19.3      2.4   0.9
#> 2 alabama,baldwin    182265  83.5  10.9      4.4   0.7
#> 3 alabama,barbour     27457  46.8  47.8      5.1   0.4
#> 4    alabama,bibb     22915  75.0  22.9      1.8   0.1
#> 5  alabama,blount     57322  88.9   2.5      8.1   0.2
#> 6 alabama,bullock     10914  21.9  71.0      7.1   0.2
```

14.2.12.2 helpers.R

`helpers.R` is an R script that can help you make choropleth maps (https://en.wikipedia.org/wiki/Choropleth_map), like the ones pictured in Figure 14.19. A choropleth map is a map that uses color to display the regional variation of a variable. In our case, `helpers.R` will create *percent_map()*, a function designed to map the data in `counties.rds`. You can download `helpers.R` from https://shiny.rstudio.com/tutorial/written-tutorial/lesson5/census-app/helpers.R.

`helpers.R` uses the **maps** and **mapproj** packages in R. If you've never installed these packages before, you'll need to do so before you make this app. Run the following command:

```
install.packages(c("maps", "mapproj"))
```

Save `helpers.R` inside your `census-app` directory, like depicted in Figure 14.21.

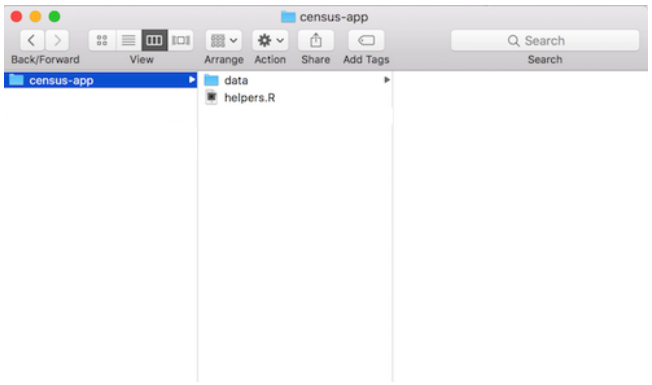


Figure 14.21: Location to store the `helpers.R` file for the US Census data app.

The *percent_map()* function in the `helpers.R` file takes five arguments:

Argument	Input
`var`	A column vector from the `counties.rds` dataset
`color`	Any character string you see in the output of *colors()*
`legend.title`	A character string to use as the title of the plot's legend
`max`	A parameter for controlling shade range (defaults to 100)
`min`	A parameter for controlling shade range (defaults to 0)

You can use the *percent_map()* function at the command line to plot the counties data as a choropleth map, like this:

```
library(maps)
library(mapproj)
source("census-app/helpers.R")
counties <- readRDS("census-app/data/counties.rds")
percent_map(var = counties$white,
            color = "darkgreen",
            legend.title = "% White")
```

> **Note: Set the correct working directory for the censusVis app.**
>
> The code above assumes that `census-app` is a sub-directory in your working directory. Make certain to set your working directory as the parent directory for `census-app`. To change your working directory location, click on **Session** → **Set Working Directory** → **Choose Directory**... in the RStudio menu bar.

The *percent_map()* function plots the counties data as a choropleth map. Here it will plot the percent of white residents in the counties in the color dark green as displayed in Figure 14.22.

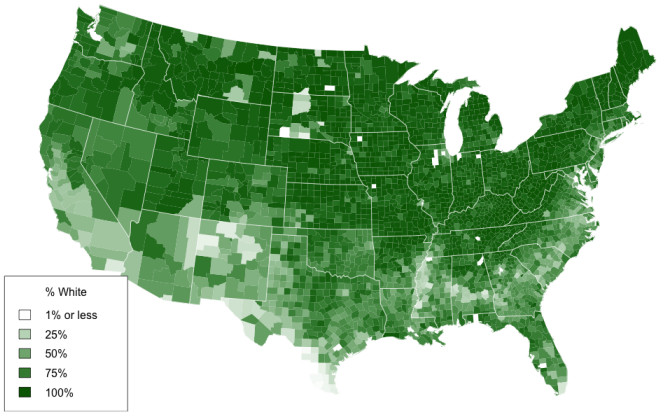


Figure 14.22: Choropleth map of the percentage white residents in the counties of the USA.

14.2.12.3 Loading files and file paths

Take a look at the above code about using the *percent_map()* function on the command line. To use *percent_map()*, we first ran `helpers.R` with the *source()* function, and then loaded `counties.rds` with the *readRDS()* function. We also ran `library(maps)` and `library(mapproj)`.

You will need to ask Shiny to call the same functions before it uses *percent_map()* in your app, but how you write these functions will change. Both *source()* and *readRDS()* require a file path, and file paths do not behave the same way in a Shiny app as they do at the command line.

When Shiny runs the commands in `server.R`, it will treat all file paths as if they begin in the same directory as `server.R`. In other words, the directory that you save `server.R` in will become the working directory of your Shiny app.

Since you saved `helpers.R` in the same directory as `server.R`, you can ask Shiny to load it with:

```
source("helpers.R")
```

Since you saved `counties.rds` in a sub-directory (named `data`) of the directory, that `server.R` is in, you can load it with:

```
counties <- readRDS("data/counties.rds")
```

You can load the **maps** and **mapproj** packages in the normal way with:

```
library(maps)
library(mapproj)
```

which does not require a file path.

14.2.12.4 Execution

Shiny will execute all of these commands if you place them in your `app.R` script. However, where you place them will determine how many times they are run (or re-run), which will in turn affect the performance of

your app, since Shiny will run some sections your `app.R` script more often than others.

Shiny will run the whole script the first time you call the *runApp()* function as displayed in Figure 14.23. This causes Shiny to execute the *server()* function.

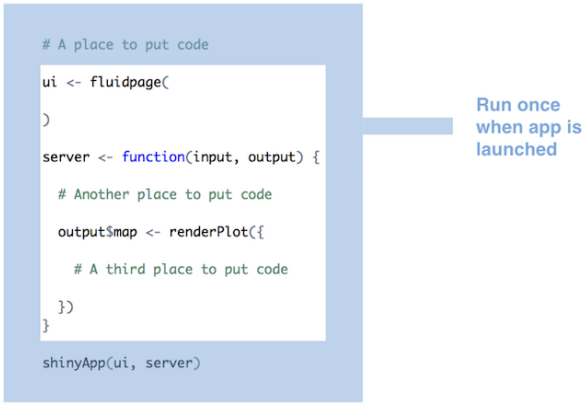


Figure 14.23: Executed part on startup of App.

Shiny saves the *server()* function until a new user arrives. Each time a new user visits your app, Shiny runs the *server()* function again, one time (see Figure 14.24). The function helps Shiny build a distinct set of reactive objects for each user.

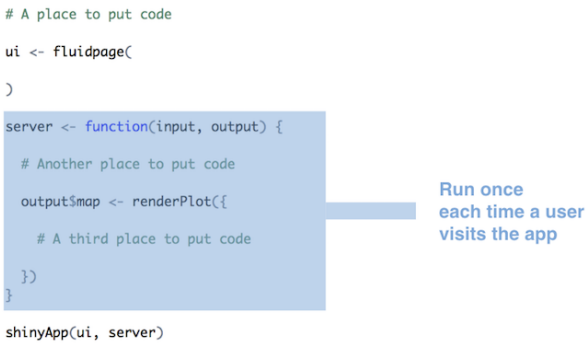


Figure 14.24: Executed part of loaded App per user visit.

As users interact with the widgets and change their values, Shiny will re-run the R expressions assigned to each reactive object that depend on a widget whose value was changed. If your user is very active, these expressions may be re-run many, many times a second (see Figure 14.25).

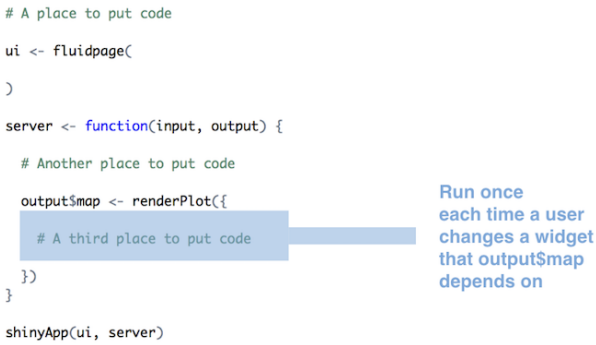


Figure 14.25: Executed part of loaded App per user widget interaction.

Here's an small overview of what you have learned so far:

- The *shinyApp()* function is run once, when you launch your app.
- The *server()* function is run once *each time* a user visits your app.

- The R expressions inside *render*()* functions are run many times. Shiny runs them once each time a user changes the value of a widget.

How can you use this information?

Source scripts, load libraries, and read data sets at the beginning of `app.R` *outside* of the *server()* function. Shiny will only run this code once, which is all you need to set your server up to run the R expressions contained in *server()*.

Define user specific objects inside the *server()* function, but outside of any *render*()* function calls. These would be objects that you think each user will need their own personal copy of. For example, an object that records the user's session information. This code will be run once per user.

Only place code that Shiny *must* rerun to build an object inside of a *render*()* function. Shiny will rerun *all* of the code in a *render*()* chunk each time a user changes a widget mentioned in the chunk. This can be quite often.

You should generally avoid placing code inside a *render*()* function that does not need to be there. Doing so will slow down the entire app.

14.2.12.5 Finishing the App

The census visualization app has one reactive object, a plot named `"map"`. The plot is built with the *percent_map()* function, which takes five arguments:

- The first three arguments, `var`, `color`, and `legend.title`, depend on the value of the select box widget.
- The last two arguments, `max` and `min`, should be the max and min values of the slider bar widget.

The *server()* function below shows one way to craft reactive arguments for *percent_map()*. R's *switch()* function can transform the output of a select box widget to whatever you like. However, the script is incomplete. It does not provide values for `color`, `legend.title`, `max`, or `min`. **Note:** the script will not run as is. You will need to finish the script before you run it, which is the task of Section 14.2.13 Exercise Shiny: Use R scripts and Data.

```
server <- function(input, output) {
  output$map <- renderPlot({
    data <- switch(input$var,
                   "Percent White" = counties$white,
                   "Percent Black" = counties$black,
                   "Percent Hispanic" = counties$hispanic,
                   "Percent Asian" = counties$asian)
    percent_map(var = data, color = ?, legend.title = ?, max = ?, min = ?)
  })
}
```

14.2.13 Exercise Shiny: Use R scripts and Data

a. Copy and paste the following `app.R` file to your census-app directory. Then add

```
source("helpers.R")
counties <- readRDS("data/counties.rds")
library(maps)
library(mapproj)
```

to your `app.R` script. Be sure to place the commands in an efficient location.

> **Note**
>
> This is the first of two steps that will complete your app. Choose the best place to insert the code above, but do not try to run the app. Your app will return an error until you replace `# some arguments` with real code in the second part of this exercise.

`ui.R`

```
# User interface ----
ui <- fluidPage(
  titlePanel("censusVis"),
  sidebarLayout(
    sidebarPanel(
      helpText("Create demographic maps with
        information from the 2010 US Census."),
      selectInput("var",
                  label = "Choose a variable to display",
                  choices = c("Percent White",
                              "Percent Black",
                              "Percent Hispanic",
                              "Percent Asian"),
                  selected = "Percent White"),
      sliderInput(inputId = "range",
                  label = "Range of interest:",
                  min = 0,
                  max = 100,
                  value = c(0, 100))
    ),
    mainPanel(plotOutput(outputId = "map"))
  )
)
# Server logic ----
server <- function(input, output) {
  output$map <- renderPlot({
    percent_map( # some arguments )
  })
}
# Run app ----
shinyApp(ui, server)
```

b. Complete the code to build a working census visualization app. When you're ready to deploy your app, save your `app.R` file and run `runApp("census-app")`. If everything works, your app should look like the picture depicted in Figure 14.26. You'll need to decide:

- how to create the argument values for *percent_map()*, and
- where to put the code that creates these arguments.

Remember, you'll want the argument values to switch whenever a user changes the associated widget.

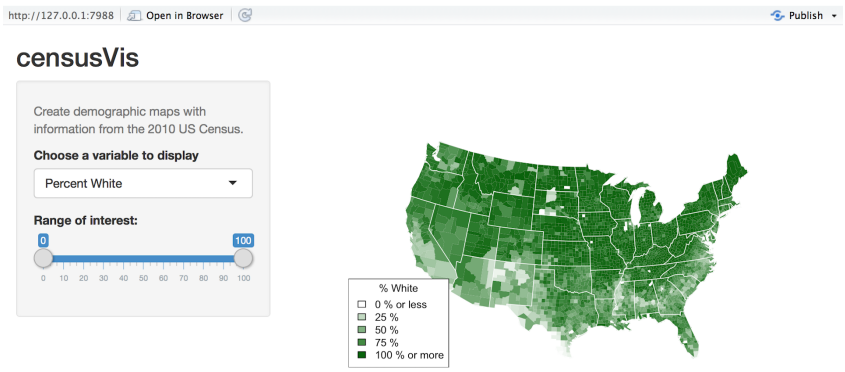


Figure 14.26: Completed US census data visualization app.

14.2.14 Use reactive expressions

Shiny apps wow your users by running fast, instantly fast. But what if your app needs to do a lot of slow computation?

This section will show you how to streamline your Shiny apps with reactive expressions. Reactive expressions let you control which parts of your app update when, which prevents unnecessary computation that can slow down your app.

To get started:

- Create a new folder named `stockVis` in your working directory.
- Download the following files and place them inside the `stockVis` directory:
  - `app.R`, download from:
    https://shiny.rstudio.com/tutorial/written-tutorial/lesson6/stockVis/app.R
  - `helpers.R`, download from:
    https://shiny.rstudio.com/tutorial/written-tutorial/lesson6/stockVis/helpers.R
- Launch the app with `runApp("stockVis")`

The stockVis App uses R's **quantmod** package, so you'll need to install **quantmod** with `install.packages("quantmod")` if you do not already have it.

14.2.14.1 A new app: stockVis

The stockVis app, as displayed in Figure 14.27, looks up stock prices by ticker symbol and displays the results as a line chart. The app lets you:

1. Select a stock to examine
2. Pick a range of dates to review
3. Choose whether to plot stock prices or the log of the stock prices on the y axis, and
4. Decide whether or not to correct prices for inflation.

Figure 14.27: Screenshot of the stockVis app user interface.

Note that the "Adjust prices for inflation" check box does not work yet. One of your tasks in this section is to fix this check box.

By default, stockVis displays the SPY ticker (an index of the entire S&P 500). To look up a different stock, type in a stock symbol that Yahoo finance will recognize. You can look up Yahoo's stock symbols from https://finance.yahoo.com/lookup/. Some common symbols are GOOG (Google), AAPL (Apple), and GS (Goldman Sachs).

stockVis relies heavily on two functions from the **quantmod** package:

1. It uses the *getSymbols()* function to download financial data straight into R from websites like Yahoo finance and the Federal Reserve Bank of St. Louis.

2. It uses the *chartSeries()* function to display prices in an attractive chart.

stockVis also relies on an R script named `helpers.R`, which contains a function that adjusts stock prices for inflation.

14.2.14.2 Check boxes and date ranges

The stockVis app uses a few new widgets:

- a date range selector, created with *dateRangeInput()*, and
- a couple of check boxes made with *checkboxInput()*. Check box widgets are very simple. They return a `TRUE` when the check box is checked, and a `FALSE` when the check box is not checked.

The check boxes are named `log` and `adjust` in the ***ui*** object, which means you can look them up as `input$log` and `input$adjust` in the *server()* function. If you'd like to review how to use widgets and their values, check out Section 14.2.8 and Section 14.2.10.

14.2.14.3 Streamline computation

The stockVis app has a problem.

Examine what will happen when you click "Plot y axis on the log scale." The value of `input$log` will change, which will cause the entire expression in the *renderPlot()* function to re-run:

```
output$plot <- renderPlot({
  data <- getSymbols(input$symb,
                     src = "yahoo",
                     from = input$dates[1],
                     to = input$dates[2],
                     auto.assign = FALSE)

  chartSeries(data,
              theme = chartTheme("white"),
              type = "line",
              log.scale = input$log,
              TA = NULL)
})
```

Each time renderPlot re-runs

1. it re-fetches the data from Yahoo finance with *getSymbols()*, and
2. it re-draws the chart with the correct axis.

This is not good, because you do not need to re-fetch the data to re-draw the plot. In fact, Yahoo finance will cut you off if you re-fetch your data too often (because you begin to look like a bot). But more importantly, re-running *getSymbols()* is unnecessary work, which can slow down your app and consume server bandwidth.

14.2.14.4 Reactive expressions

You can limit what gets re-run during a reaction with reactive expressions.

A reactive expression is an R expression that uses widget input and returns a value. The reactive expression will update this value whenever the original widget changes.

To create a reactive expression use the *reactive()* function, which takes an R expression surrounded by braces (just like the *render*()* functions).

For example, here's a reactive expression that uses the widgets of stockVis to fetch data from Yahoo:

```
dataInput <- reactive({
  getSymbols(input$symb,
             src = "yahoo",
             from = input$dates[1],
             to = input$dates[2],
```

```
              auto.assign = FALSE)
})
```

When you run the expression, it will run *getSymbols()* and return the results, a data frame of price data. You can use the expression to access price data in *renderPlot()* by calling *dataInput()*, as shown here:

```
output$plot <- renderPlot({
  chartSeries(dataInput(),
              theme = chartTheme("white"),
              type = "line",
              log.scale = input$log,
              TA = NULL)
})
```

Reactive expressions are a bit smarter than regular R functions. They cache their values and know when their values have become outdated. What does this mean? The first time that you run a reactive expression, the expression will save its result in your computer's memory. The next time you call the reactive expression, it can return this saved result without doing any computation (which will make your app faster).

The reactive expression will only return the saved result if it knows that the result is up-to-date. If the reactive expression has learned that the result is obsolete (because a widget has changed), the expression will recalculate the result. It then returns the new result and saves a new copy. The reactive expression will use this new copy until it too becomes out of date.

To summarize this behavior:

- A reactive expression saves its result the first time you run it.
- The next time the reactive expression is called, it checks if the saved value has become out of date (i.e., whether the widgets it depends on have changed).
- If the value is out of date, the reactive object will recalculate it (and then save the new result).
- If the value is up-to-date, the reactive expression will return the saved value without doing any computation.

You can use this behavior to prevent Shiny from re-running code unnecessarily. Consider how a reactive expression will work in the new stockVis app below.

```
server <- function(input, output) {

  dataInput <- reactive({
    getSymbols(input$symb,
               src = "yahoo",
               from = input$dates[1],
               to = input$dates[2],
               auto.assign = FALSE)
  })

  output$plot <- renderPlot({

    chartSeries(dataInput(),
                theme = chartTheme("white"),
                type = "line",
                log.scale = input$log,
                TA = NULL)
  })

}
```

When you click "Plot y axis on the log scale", `input$log` will change and *renderPlot()* will re-execute. Now

1. *renderPlot()* will call *dataInput()*,
2. *dataInput()* will check that the `dates` and `symb` widgets have not changed,

3. *dataInput()* will return its saved data set of stock prices *without re-fetching data from Yahoo*,
4. *renderPlot()* will re-draw the chart with the correct axis.

14.2.14.5 Dependencies

What if your user changes the stock symbol in the `symb` widget?

This will make the plot drawn by *renderPlot()* out of date, but *renderPlot()* no longer calls `input$symb`. Will Shiny know that `input$symb` has made the plot out of date?

Yes, Shiny will know and will redraw the plot. Shiny keeps track of which reactive expressions an ***output*** object depends on, as well as which widget inputs. Shiny will automatically re-build an object if

- an ***input*** value in the objects's *render*()* function changes, or
- a reactive expression in the objects's *render*()* function becomes obsolete

Think of reactive expressions as links in a chain that connect ***input*** values to ***output*** objects. The objects in ***output*** will respond to changes made anywhere downstream in the chain. (You can fashion a long chain because reactive expressions can call other reactive expressions.)

Only call a reactive expression from within a *reactive()* or a *render*()* function. Why?

Only these R functions are equipped to deal with reactive output, which can change without warning. In fact, Shiny will prevent you from calling reactive expressions outside of these functions.

14.2.15 Exercise Shiny: Fixing the stockVis app

a. Time to fix the broken check box for "Adjust prices for inflation." Your user should be able to toggle between prices adjusted for inflation and prices that have not been adjusted.

The *adjust()* function in `helpers.R` uses the Consumer Price Index data (http://research.stlouisfed.org/fred2/series/CPIAUCNS) provided by the Federal Reserve Bank of St. Louis to transform historical prices into present day values. But how can you implement this in the app?

Here's one solution below, but it is not ideal. Can you spot why? Once again it has to do with `input$log`.

```
server <- function(input, output) {

  dataInput <- reactive({
    getSymbols(input$symb,
               src = "yahoo",
               from = input$dates[1],
               to = input$dates[2],
               auto.assign = FALSE)
  })

  output$plot <- renderPlot({
    data <- dataInput()
    if (input$adjust) data <- adjust(dataInput())

    chartSeries(data,
                theme = chartTheme("white"),
                type = "line",
                log.scale = input$log,
                TA = NULL)
  })
}
```

b. Fix this problem by adding a new reactive expression to the app. The reactive expression should take the value of *dataInput()* and return an adjusted (or not adjusted) copy of the data. When you think you have it, compare your solution to the model answer on Brightspace. Make sure you understand what calculations will happen and what calculations will not happen in your app when your user clicks "Plot y axis on the log scale".

14.2.16 Share your apps

You can now build a useful Shiny app, but can you share it with others? This section will show you several ways to share your Shiny apps.

When it comes to sharing Shiny apps, you have two basic options:

Share your Shiny app as R scripts. This is the simplest way to share an app, but it works only if your users have R on their own computer (and know how to use it). Users can use these scripts to launch the app from their own R session, just like you've been launching the apps so far.

Share your Shiny app as a web page. This is definitely the most user-friendly way to share a Shiny app. Your users can navigate to your app through the internet with a web browser. They will find your app fully rendered, up to date, and ready to go.

14.2.16.1 Share as R scripts

Anyone with R can run your Shiny app. They will need a copy of your `app.R` file, as well as any supplementary materials used in your app (*e.g.*, `www` folders or `helpers.R` files).

To send your files to another user, email the files (perhaps in a zip file) or host the files online.

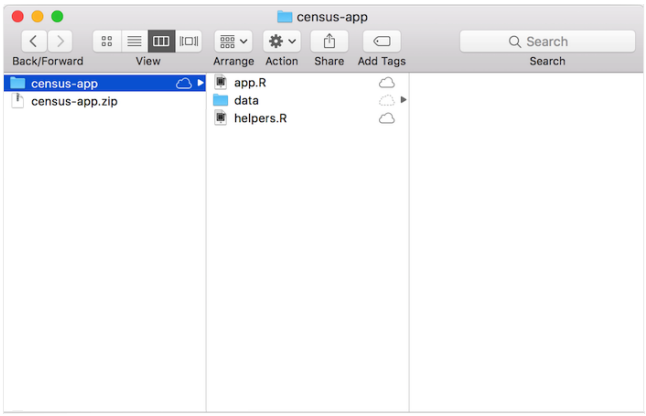


Figure 14.28: App received as zip file and extracted.

Your user can place the files into an app directory in their working directory as shown in Figure 14.28. They can launch the app in R with the same commands you used on your computer:

```
# install.packages("shiny")
library(shiny)
runApp("census-app")
```

Shiny has three built in commands that make it easy to use files that are hosted online: *runUrl()*, *runGitHub()*, and *runGist()*.

runUrl *runUrl()* will download and launch a Shiny app straight from a web-link.

To use *runURL()*:

- Save your Shiny app's directory as a zip file
- Host that zip file at its own link on a web page. Anyone with access to the link can launch the app from inside R by running:

```
library(shiny)
runUrl( "<the weblink>")
```

runGitHub If you don't have your own web page to host the files at, you can host your the files for free at https://github.com/.

GitHub is a popular project hosting site for R developers since it does more than just host files. GitHub provides many features to support collaboration, such as issue trackers, wikis, and close integration with the git (https://git-scm.com/) version control system. To use GitHub, you'll need to sign up (it's free) and choose a user name.

To share an app through GitHub, create a project repository on GitHub. Then store your `app.R` file in the repository, along with any supplementary files that the app uses.

Your users can launch your app by running:

```
runGitHub( "<your repository name>", "<your user name>")
```

runGist If you want an anonymous way to post files online, GitHub offers a pasteboard service for sharing files at https://gist.github.com/. You don't need to sign up for a GitHub account to use this service. Even if you have a GitHub account, gist can be a simple, quick way to share Shiny projects.

To share your app as a gist:

- Copy and paste your `app.R` files to the gist web page.
- Note the URL that GitHub gives the gist.

Once you've made a gist, your users can launch the app with `runGist("<gist number>")` where `"<gist number>"` is the number that appears at the end of your Gist's web address.

Here (https://gist.github.com/mine-cetinkaya-rundel/eb3470beb1c0252bd0289cbc89bcf36f) is an example of an app hosted as a gist. You could launch this app with:

```
runGist("eb3470beb1c0252bd0289cbc89bcf36f")
```

14.2.16.2 Share as a web page

All of the above methods share the same limitation. They require your user to have R and Shiny installed on their computer.

However, Shiny creates the perfect opportunity to share output with people who do *not* have R (and have no intention of getting it). Your Shiny app happens to be one of the most widely used communication tools in the world: a web page. If you host the app at its own URL, users can visit the app (and not need to worry about the code that generates it).

If you are familiar with web hosting or have access to an IT department, you can host your Shiny apps yourself.

If you'd prefer an easier experience or need support, RStudio offers four ways to host your Shiny app as a web page:

1. shinyapps.io
2. Shiny Server
3. Posit Connect

Shinyapps.io The easiest way to turn your Shiny app into a web page is to use https://www.shinyapps.io/, RStudio's hosting service for Shiny apps.

shinyapps.io lets you upload your app straight from your R session to a server hosted by RStudio. You have complete control over your app including server administration tools. You can find out more about shinyapps.io by visiting shinyapps.io.

Shiny Server Shiny Server, see https://posit.co/download/shiny-server/, is a companion program to Shiny that builds a web server designed to host Shiny apps. It's free, open source, and available from GitHub.

Shiny Server is a server program that Linux servers can run to host a Shiny app as a web page. To use Shiny Server, you'll need a Linux server (64 bit) that has explicit support for Ubuntu, CentOS/RHEL 7+, and openSUSE/SLES Enterprise Linux 12. If you are not using an explicitly supported distribution, you can still use Shiny Server by building it from source.

You can host multiple Shiny applications on multiple web pages with the same Shiny Server, and you can deploy the apps from behind a firewall.

To see detailed instructions for installing and configuring a Shiny Server, visit the Shiny Server guide (https://github.com/rstudio/shiny-server/blob/master/README.md).

Posit Connect Posit Connect is a publishing platform for the work your teams create in R. Share Shiny applications, R Markdown reports, dashboards, plots, and more in one convenient place. With RStudio

Connect, you can publish from the RStudio IDE with the push of a button and schedule execution of reports and flexible security policies.

If you'd like to learn more about RStudio Connect and the features it offers go to https://posit.co/products/enterprise/connect/.

Lesson 15

End Assignment of the Course MAT27803

The end assignment of the Course MAT27803 "R for Statistics" will be made available via Brightspace. On the assignment you will find instructions on how, and before which date you have to submit your answers.

Bibliography

[1] R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, 2022. https://www.R-project.org/.

[2] J.M. Chambers, Programming with data: A guide to the s language, 1st ed., Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[3] RStudio Team, RStudio: Integrated Development Environment for R, RStudio, PBC., Boston, MA, 2022. http://www.rstudio.com/.

[4] R.L. Ott, M. Longnecker, An introduction to statistical methods and data analysis, 6^{th} International Student Edition, Brooks/Cole, Cengage Learning, 2010.

[5] R.L. Ott, M. Longnecker, An introduction to statistical methods & data analysis, 7^{th} Edition, Cengage Learning, 20 Channel Center Street, Boston, MA 0210, USA, 2016.

[6] J. Fox, S. Weisberg, An R companion to applied regression, Second, Sage, Thousand Oaks, CA, USA, 2011. http://socserv.socsci.mcmaster.ca/jfox/Books/Companion.

[7] Y. Xie, Knitr: A general-purpose package for dynamic report generation in r, 2022. http://yihui.name/knitr/.

[8] Y. Xie, Dynamic documents with R and knitr, 2nd ed., Chapman; Hall/CRC, Boca Raton, Florida, 2015. http://yihui.name/knitr/.

[9] Y. Xie, Knitr: A comprehensive tool for reproducible research in R, in: V. Stodden, F. Leisch, R.D. Peng (Eds.), Implementing Reproducible Computational Research, Chapman; Hall/CRC, 2014. http://www.crcpress.com/product/isbn/9781466561595.

[10] Y. Xie, formatR: Format r code automatically, 2022. https://CRAN.R-project.org/package=formatR.

[11] J. Hester, F. Angly, R. Hyde, M. Chirico, K. Ren, A. Rosenstock, I. Patil, Lintr: A 'linter' for r code, 2022. https://CRAN.R-project.org/package=lintr.

[12] A. Canty, B.D. Ripley, Boot: Bootstrap r (s-plus) functions, 2021.

[13] A.C. Davison, D.V. Hinkley, Bootstrap methods and their applications, Cambridge University Press, Cambridge, 1997. http://statwww.epfl.ch/davison/BMA/.

[14] M.B. Brown, A.B. Forsythe, Robust tests for the equality of variances, Journal of the American Statistical Association 69 (1974) 364–367. http://www.jstor.org/stable/2285659.

[15] J.M. Chambers, Programming with data: A guide to the S language, Springer Science & Business Media, 1998.

[16] H. Wickham, The Split-Apply-Combine Strategy for Data Analysis, Journal of Statistical Software 40 (2011) 1–29. https://doi.org/10.18637/jss.v040.i01.

[17] G.E.P. Box, Problems in the Analysis of Growth and Wear Curves, Biometrics 6 (1950) 362–389. https://doi.org/10.2307/3001781.

[18] P. Murrell, R Graphics, 2nd ed., CRC Press, Inc., Boca Raton, FL, USA, 2011.

[19] D. Sarkar, Lattice: Multivariate Data Visualization with R, Springer, New York, 2008. https://doi.org/10.1007/978-0-387-75969-2.

[20] H. Wickham, ggplot2: Elegant graphics for data analysis, Springer-Verlag New York, 2016. https://doi.org/10.1007/978-3-319-24277-4.

[21] W. Chang, H. Wickham, ggvis: Interactive Grammar of Graphics, 2016. https://CRAN.R-project.org/package=ggvis.

[22] W.N. Venables, D.M. Smith, R. Gentleman, R. Ihaka, M. Maechler, the R Core Team, An introduction to R: Notes on R: A Programming Environment for Data Analysis and Graphics, 2022. https://cran.r-project.org/manuals.html.

[23] H. Wickham, Tidy Data, Journal of Statistical Software 59 (2014) 1–23. https://doi.org/10.18637/jss.v059.i10.

[24] S.J. Welham, S.A. Gezan, S.J. Clark, A. Mead, Principles for designing experiments, in: Statistical Methods in Biology: Design and Analysis of Experiments and Regression, Chapman and Hall/CRC Press, Boca Raton, FL USA, 2014: pp. 43–68.

[25] J. Cohen, ed., Statistical power analysis for the behavioral sciences, 2nd ed., Lawrence Erlbaum Associates, 1988.

[26] W. Chang, J. Cheng, J. Allaire, C. Sievert, B. Schloerke, Y. Xie, J. Allen, J. McPherson, A. Dipert, B. Borges, shiny: Web Application Framework for R, 2022. https://CRAN.R-project.org/package=shiny.

Index