Five days of R

A crash-course in R

January Weiner

Invalid Date

Table of contents

Introduction

Five days of R

I have been teaching R to biologists and medical students for many years now. At the Core Unit for Bioinformatics at the Berlin Institute of Health, Charité - Universitätsmedizin Berlin, we have developed a five-day, 5 hour per day R crash course running for the last three years. This book is a companion to that course.

This is also the reason for how the materials in the book are arranged. Rather then discussing everything about vectors first, then everything about matrices etc., we start with easy things, and return to them later to build on them. I call this "helical learning"[1] – we spiral around the same topics, but each time going a bit deeper, and each time you will understand a bit more. This is also why some topics are spread between the days – by trial and error, we have found the amount of material that can be covered in a day of learning.

[1] I got this idea from the professor Barbara Płytycz from the Jagiellonian University, who taught me my first "helix" of the immune system.

There are two goals of this book. The first one is that after five days of learning R, you will be able to load, inspect, manipulate and save data files (such as Excel tables or CSV files), make some basic plots and perform simple statistical tests. The second goal is that you are in a good starting position to continue learning R on your own.

In other words, this course should give you a jump start, allowing to overcome this first big hurdle in learning R.

Prerequisites

Installing R and RStudio

Before you dive in to the course, we would like to ask you to install both R and RStudio: you need R (obviously) and RStudio is a great and very popular interface for R. Minimal installation guide:

1. Install R. Go to R website and download the version for your operating system. Install it.
2. Install R Studio. Go to R Studio installation page and follow the instructions.

This should be it. However, if you are stuck or have problems with installing R or RStudio, please search the internet – the installation and especially related problems unfortunately depends on your operating system and the version of the software, so it is hard to give a general advice.

Here are some other websites with somewhat more detailed instructions:

- Installing R on all systems
- Installing R on Windows 7, 8, 10
- Installing R, RStudio and configuring RStudio
- Getting started chapter of the R Cookbook by James JD Long.

Potential problems:

1. **You already had R installed on the system.** Having multiple R versions may be a source of problems. If you have an old version of R, unless you have a good reason to keep it, uninstall it.
2. **You need to compile packages.** Packages come in two forms: precompiled packages and source packages. Some exotic packages might not have the precompiled version for your system. If you need to compile packages, you will have to install additional packages. For example, for Windows, you need to install Rtools - make sure you

download the version that corresponds to your version of R.

Installing R packages

During the course we will use several R packages that you need to install on your computer. On Day 2, we will discuss installing and loading packages, and we will make a note to install the required packages when they are needed. However, you can also install them right now, after you have installed R and RStudio. This can be more effective – after all, installing might take some time.

Here is the list of packages that you will have to install:

- `tidyverse`
- `ggplot2`
- `skimr`
- `pander`
- `readxl`
- `writexl`
- `janitor`
- `broom`
- `cowplot`
- `ggbeeswarm`
- `pheatmap`
- `tinytex` (only if you want to produce PDF output from Rmarkdown)

You can install them by running the following code in your RStudio:

```
install.packages(c("tidyverse", "ggplot2", "skimr", "pander",
                   "readxl", "writexl", "janitor", "broom",
                   "cowplot", "ggbeeswarm", "pheatmap", "tinytex"))
```

The structure of this book

This book is divided into five chapters, each corresponding to one day of the course. At the beginning of each chapter, you

will find a short list of topics for the given day.

Some parts of the book are highlighted:

Code blocks with output:

```
# this is a comment
x <- 1 + 1
```

The numbers on the left (if present) are not part of the code – they are just line numbers. You can copy the code by clicking on the "Copy" button ( ) and it will not copy these numbers.

Exercise 0.1 (Example)**.**

- This is how an exercise looks like!
- Please do all exercises. It helps a lot.
- Some things are learned only through exercises.

Solution (click me!)

Some exercises have a solution which you can click to reveal.

Useful tips

- Some exercises have solutions in the "Solutions" chapter. If they do, please read the solution after you have completed the exercise – often there will be a comment or a hint that will help you understand the material.

Remember!

- Run *all* code chunks in your RStudio.
- Do *all* exercises.
- Go through the "Review" section at the end of each chapter and make sure you understand *everything* on the list.

Take a look at the right margin!

New concepts are highlighted on the right margin of the book

And again[2]!

[2] And also footnotes.

In each chapter there are several exercises. *However*, that does not mean that you should *only* do the exercises. In fact, you should try out *every* piece of code that is in the book. Copy it (there is a button next to each code block that will do it for you), paste it into your RStudio and run it. Then try to modify it and see what happens.

Exercises in this book are important. They are not only there to check if you understood the material, but they can also introduce new concepts or ideas. This is because this book is not only about learning R, but also learning *how to learn about R*. So, for example, sometimes we will want you to figure stuff on your own rather than give you a ready-made answer.

Many of the exercises have a solution provided, either inline (you have to expand it by clicking on the "Solution" button) or in the "Solutions" section of the book. **It is really important that you do not give up too quickly**. Try to solve the exercise on your own, and only then look at the solution.

Each chapter is ended by a "Review" section, which contains a list of things that you have learned that day. It is really important that you go through that list and make sure that you understand everything on it. Some of the new things appeared in the exercises, so if you skipped them, you might want to go back and do them.

If you do all that, I personally *guarantee you* that by the end of this course you will be able to use R in your work.

General advice

The course is a real crash course. There is a lot of material coming at you in a very short time. You will feel overloaded and overwhelmed – this is normal. Don't worry! It will soon get better, and in a few days you will be able to do fairly advanced things with R.

The key is to keep playing with your R; trying out new things, breaking it. Please go through all exercises in that book, even

if they seem simple at the first glance (some of them are tricky, others are used to smuggle in new concepts and useful tidbits of information).

Whenever you feel you don't understand something, stop and try to figure it out. Use internet search very liberally. Many answers can be found on sites such as StackOverflow, R-bloggers, or simply in the R documentation[3]. Try out the code you will find in these sources – just copy-paste it into your RStudio and adapt it to your needs. Feel free to use Large Language Models (such as GPT) – they are very good at explaining code, especially when you are learning basic concepts.

[3] You can access the R documentation by typing `?function_name` in the console. You can search for concepts using `??keyword`.

However, if you want to learn R, simply doing this course will not be enough. You need to start using it in a real world setting. Unfortunately – the better you already are at Excel, Word and other such tools, the harder it will be switching to R: tasks that are a breeze in Excel will at first require you to spend substantially more time in R. However, trust me: it pays off in the long run. Therefore, for best results, force yourself to use R *even if at first it is less efficient then other tools.*

Finally: programming can be fun. Most programmers I know simply enjoy doing that. It is a satisfaction similar to building Lego models or solving puzzles or reading a crime story – and also very much like doing experiments. Unlike experiments, however, you can't break expensive equipment and the results of your attempts are usually immediately visible, no need to wait for weeks for the results. You can play, experiment, try out various things at no expense and no risk. If you can get into this mindset, learning R will be much, much easier.

Acknowledgements

I have been teaching statistics and bioinformatics for more than two decades now. About ten years ago, I started teaching the first "R crash course" with the goal of introducing PhD students and postdocs to R as quickly and as painlessly as possible. The course evolved over the years, and I had many partners and co-teachers.

In the first place, I would like to thank Carlo Pecoraro from Physalia Courses for the opportunity to teach my first R crash course for Physalia.

Several times I have taught the course together with my colleague, Dr. Manuela Benary from the Core Unit of Bioinformatics at the Berlin Institute of Health, Charité - Universitätsmedizin Berlin. Manuela has been a great partner in teaching the course, and many ideas in this book are actually hers.

I would also like to thank many colleagues and friends who had the patience to go through this book and provide feedback.

Finally, I thank you, the Reader, for your interest in this book. I hope it will be useful and that you will enjoy using R in your research.

1 First day of R

1.1 Goals for today

- what is R?
- why use R?
- first steps in R

What you should know after today:

- what R is
- how to start R
- how to use R as a calculator
- how to assign variables
- how to use functions
- how to use vectors
- how to use data frames
- how to use packages

1.2 R, RStudio and other languages

1.2.1 Why R?

1.2.2 R and RStudio

R is the name of both, the programming language and of the language *interpreter*. When you start RStudio, you can see the R language interpreter working in the part of the window left and bottom - called "console". So yes, you don't need RStudio to work with R and, in fact, many people prefer to work with R in a different environment.

RStudio is a so called IDE, an Integrated Development Environment. That is, it provides a lot of goodies that help make your work easier, faster and more efficient.

1.2.3 R and other languages

R is not the only language that you can use for data analysis. There are many other languages that are used for this purpose, including Python, Matlab and many others. Each of these languages has its own strengths and weaknesses, and the choice of language depends on your needs. In fact, most bioinformaticians know more than one language, and use the one that is best suited for the task at hand.

We think that R is a particularly good choice for all those who just need a tool to use from time to time to help them with their work. It is relatively easy to learn, and it is very powerful. However, other choices are also worth mentioning.

Matlab

Matlab is a language that is in many ways similar to R. The main differnce is maybe that unlike R, Matlab is not free – it is closed source and you have to pay for a license. This has some advantages. For example, and as you will see during this course, R development is not centralized and so there are many packages that do the same thing. Matlab is in some aspects more consistent and more polished than R, and in some comparisons appears to be faster – and for this, it is often the language of choice for areas such as image analysis.

Python

Python is completely different story. This is a powerful, fast, general purpose programming language. It is more versatile than R, has a much more standardized syntax and development process. However, it is harder to learn and it is not really meant to be used interactively (although it can be – especially when combined with Quarto or Jupyter Notebook). While many statistical modules exist for Python, it is not as strong in this area as R.

1.3 Projects and Workspaces

1.3.1 Creating a project: start here!

When starting work with a new project, do the following: (i) create a new directory for the project, (ii) open an R script file and save it in the directory you created and (iii) copy necessary data files.

RStudio projects

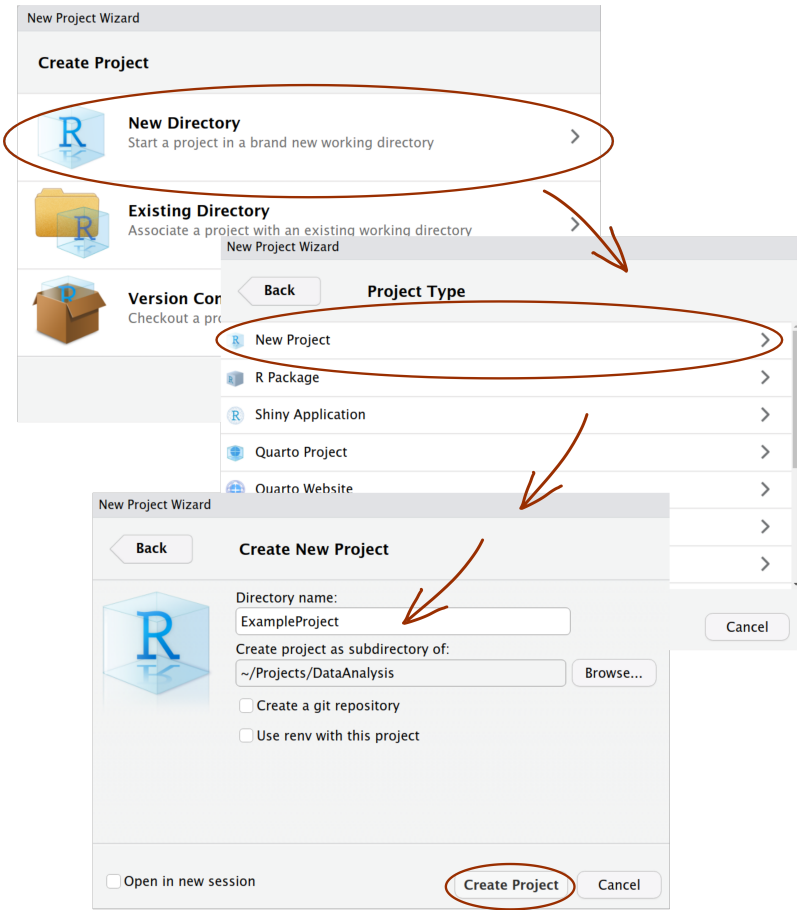


Figure 1.1: Creating a new project in RStudio

To create a new directory in RStudio, go to File -> New Project. When the dialog window appears, select first "New Directory" and then "New Project". Click on "Browse..." to select the

location where you would like to have the directory created. Enter a name for your project and click on "Create Project". Presto!

Exercise 1.1.

- Create a new project in RStudio. Do it now, you will continue to work with this project over the next few days (we hope).
- Inspect the contents of the project directory. What files are there?

When Rstudio creates a new project, it creates a new directory with the same name as the project. Furthermore, it creates a new file in this directory called `projectname.Rproj`. This file is used by RStudio to keep track of project-specific settings.

`projectname.Rproj`

You can open this file by double-clicking on it in the Files pane in RStudio. Like most of the files that you will be working with, it is a simple text file: you can open it in any text editor, including RStudio.

The other file I told you to create is a *script* file. This is where you will later be typing your code, and we will discuss it in more detail in a moment.

Later on, if you choose to do so, R can create two hidden files, `Rhistory` (called `.Rhistory` on Unix-like systems and `_Rhistory` on Windows) and `.RData` (or `_RData`). This files save the state of your R session (of your R workspace, to be specific).

`.Rhistory` and `.RData`

1.3.2 RStudio components

As I mentioned before, RStudio is a so-called "integrated development environment" (IDE). It makes working with R much easier and more efficient. You will be exploring many of its components when you start working with R and RStudio, but let us just mention a couple general things. Below is a screenshot of an R session opened in RStudio:

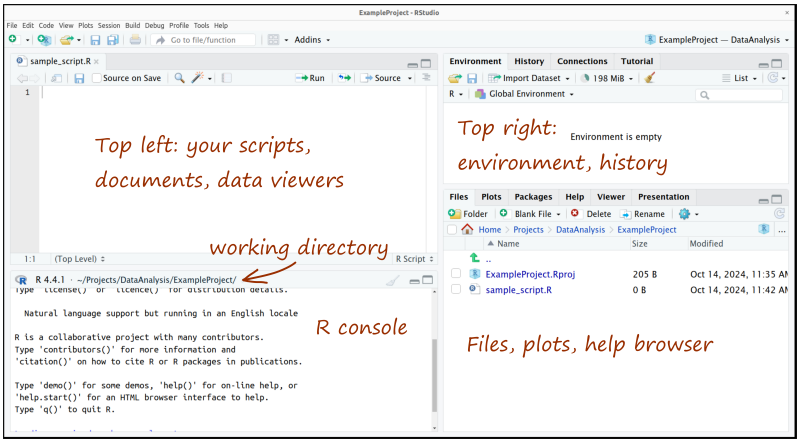


Figure 1.2: RStudio components

There are four main RStudio panels on the screenshot above[4]. **Top left** is where you usually see your scripts, R markdown documents or any other opened files. Sometimes you will have a tab with the view of a data frame. Most of the time, this is where you will be working – mostly typing your code in a script or R markdown file. Chances are that you don't see this panel yet, as you have not opened any files – you will see it in a moment.

[4] You can customize that view, of course. You can change colors, position, elements shown and much, much more. Take a look in Tools -> Global Options.

On the **top right** panel you have several tabs. You will soon be using the "Environment" tab, which shows you the variables that you have created and their values. The "History" tab shows you the history of commands[5] that you have typed in the console. There are other tabs here that might come in handy at some point in your work with R – one that you might want to try to supplement this course is the "Tutorial", which walks you through basics of R (a bit like this book).

[5] The "commands" that you execute in R are properly called "expressions".

The **bottom left** panel is where the actual R is, or, more precisely, the R console[6]. The R console is where you can directly access R. If you were to start R standalone (without R studio), the console is the only thing that you would see. You will start typing in the console in a moment, but soon you will learn to indirectly access the console by typing your code in the script and executing it. Note that at the top of the console you can

[6] Pronounced / k n.sə l/

see the path to your project directory. We will discuss paths and directories in a moment.

Finally, on the **bottom right** you have, again, several tabs. "Files" is a file browser. You can navigate your file system here, click to load files in R or preview them, and more. In another tab, you will see the plots once you generate them.

Exercise 1.2 (Help)**.** Click on the "Help" tab in the right bottom panel. Type "t.test" in the search bar (mind the dot!) and press Enter. What do you see? Go through the document and notice the structure of the manual page. The majority of the manual pages will have precisely the same structure, as it is a part of the R documentation system.

1.4 Let's start with R

1.4.1 R as a calculator

You can use R as a very powerful calculator. For example, do you want to know what $\sin(\pi/2)$ is? Just type `sin(pi/2)` in the console and press `Enter`. Addition and subtraction work, as expected, with `+` and `-`. To multiply two numbers, type `2*3`; to divide, type `2/3`. You can get exponents (powers, eg. 2^3) by typing `2^3`. If the `^` symbol (called "caret") is not available on your keyboard, you can use `**` instead. Parentheses `()` are used to group expressions, just like in mathematics. To logarithmize, you can use `log()`, `log2()` and `log10()` functions. For example, to calculate $\log_{10}(100)$, type `log10(100)`. Can you guess how to calculate $\sqrt{2}$? Yes, you are right: `sqrt(2)`. Or `2^(1/2)`, that will also do. Finally, the `exp()` function calculates the exponential function e^x.

`sin()`

`log()`, `log2()` and `log10()`

`sqrt()`

`exp()`

Exercise 1.3. Calculate the following expressions in R:

- $\log_2(8)$
- $\sin(\pi)$
- 2^{10}

- $\sqrt{e}$

Solution

```
log2(8)
```

```
[1] 3
```

```
sin(pi)
```

```
[1] 1.224647e-16
```

```
2^10
```

```
[1] 1024
```

```
sqrt(exp(1))
```

```
[1] 1.648721
```

1.4.2 Using script files

On the left side of the RStudio window you have (by default) two panels: the lower one is called "Console". When you create a new script file, as you have done it a moment ago, it appears above.

Typing in console

You *can* type your commands (properly called "expressions") directly into the console, but it is generally not a good idea. Why? The truthful answer is: because it is messy and sooner or later you will regret it. You *can* save the history of what you type in the console, but it is easier (and cleaner) to save your program in a script file.

When you open or create an R script file and type something

into it, you can send it to console and execute it. To do that, you have two options. First, you can use the "Run" button in the script panel:

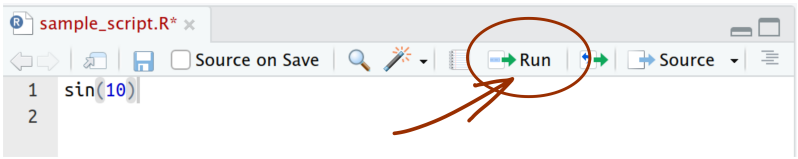


Figure 1.3: Using the "Run" button

However, this is one of the most common operations, so it is much more efficient to use a keyboard shortcut: `Ctrl+Enter` (or `Cmd+Enter` on Mac). This will send the current line of code to the console, where it will be executed, and the cursor in the script will move to the next line of code. You can also **select** a fragment of the code before you press `Ctrl+Enter`, and then the whole selected fragment will be sent to the console.

> **Keyboard shortcuts**
>
> There are many keyboard shortcuts in RStudio. You can see them all in the "Help" menu, under "Keyboard Shortcuts Help". You can also customize them in the "Tools" -> "Modify Keyboard Shortcuts" menu.

1.4.3 Comments

If you start your line with `#` (called "hash" or "pound" sign), the rest of the line will be ignored by R. This is called a **comment** and I will spend some time later on convincing you that you should use a lot of comments in your code.

Comments

```
# this is a comment
# this is another comment
x <- 2

# the following will not be executed:
# x <- 5
```

Comments are also a great way to temporarily disable a line of code - we call it "commenting out". This is for the cases when you want to try out something, but you do not want to delete a line of code that may be still useful later on.

Exercise 1.4. Repeat Exercise 1.3, but now type the expressions into the script which you have created in Exercise 1.1. Before each expression, insert a comment line stating what it does, for example `# calculate sinus of pi`. Run the script by pressing `Ctrl+Enter` after each line.

Script files are also text documents. You can open them in any text editor, for example Notepad or even Word (but don't do that). In RStudio, you see the script file in many colors: for example, comments can be green, strings (text in quotes) can be red, and so on. This is called **syntax highlighting** and is done by RStudio to make your code more readable. You will not see the colors when you open your R script in Notepad.

> **!** Important
>
> From now on, you should only type your code in script files.

1.4.4 Variables

What if we want to store the result of a calculation for later use? We can do this by assigning the result to a variable. In R, you assign a value to a variable using the `<-` operator:

`<-` assignment operator

```
x <- 2
y <- sin(pi/2)
z <- x + y
```

If you want to see the value of a variable, just type its name in the console and press `Enter`, or use `print()` function:

`print()`

```
print(z)
```

```
[1] 3
```

> Assignment operator
>
> Many other languages use `=` as an assignment operator. In R, you can use `=` as well, but do yourself a favour and **don't**. Use `<-` instead. Why? Your code will be more readable and you will avoid many common mistakes.

Variables are like boxes in which you can store values. However, unlike boxes, when you assign one variable to another, the first variable keeps its content:

```
x <- 2
y <- x
x
```

```
[1] 2
```

We now come to a very important point which we will revisit often, as it is one of the most common beginner (and not only beginner) mistakes. When you forget to assign the value to a variable, R will print it to the console, but the variable **will not be modified**:

```
x <- 2
# prints 0.9092974:
sin(x)
# prints 2
x
```

In the code above, the value of `x` is not changed by the `sin()` function. **To store the value of a function, you need to assign it to a variable**:

```
x <- 2
# does not print anything:
x <- sin(x)
# prints 0.9092974:
x
```

Please spend some time on this, as it is a very common source of errors.

Exercise 1.5. Without actually running the code, guess what will be the value of `x` if you execute the following code:

```
# first, assign starting value
x <- 2

# multiply by 100
x <- x * 100

# power of 2, add 10
x^2 + 10

# divide by 10
x / 10

# subtract 10*20 and take the square root
x <- sqrt(x - 10 * 20)
```

Solution

Just run the code ;-) Which lines contain the assignment operator?

! Important

As a rule of thumb[1], if the expression you type in your script does not contain the `<-` operator, it will not modify any variables.

Exercise 1.6. Create a variable using `x <- 42`. Take a look at the Environment pane in RStudio (top left part of the window). Do you notice anything?

[1] There are exceptions to this rule, but they are relatively rare and I will not discuss them here.

Solution

A new entry appeared in the "Environment pane". It shows that there is a new variable present in your environment.

1.4.5 Character variables

Variables can store not only numbers, but also text. Text in R is called a **character string**. To create a character string, you need to enclose the text in quotes (both single and double quotes are allowed, but try to be consistent and use only one type). For example:

```
name <- "January"
city <- "Hoppegarten"
greeting <- "Hello, world!"
```

Character variables cannot be used with algebraic computations, the following code will throw an error:

```
# this does not work!
name + city
```

```
Error in name + city: non-numeric argument to binary operator
```

However, if you want to "add" two character strings (that is, concatenate them), you can use the `paste()` function: `paste()`

```
paste(name, city)
```

```
[1] "January Hoppegarten"
```

Quite often, you don't want to have a space between the two strings. This is such a common operation that R has a shortcut for it: `paste0()`

```r
paste0(name, city)
```

```
[1] "JanuaryHoppegarten"
```

Other types

There are other types of data types in R. Later on, we will briefly touch on *factors*, which look like character strings but behave like numbers. Another important data type is a *logical* type, which can have only two values: `TRUE` and `FALSE`. We will talk about logical types in more detail tomorrow. And under the hood, numeric vectors can be either integers (numbers like 1, 2, …) or floating point numbers (numbers like 1.1, 2.2 or π).

Exercise 1.7 (Variables)**.** Create the following variables in your script:

- `name` with the value of your first name
- `city` with the value of the city where you live
- `age` with the value of your age
- `greeting` with the value "Hello,"
- concatenate the variables `greeting` and `name` and store the result in a new variable `hellothere`

Solution

```r
name <- "January"
city <- "Hoppegarten"
age <- 199
greeting <- "Hello,"
hellothere <- paste(greeting, name)
```

1.4.6 Workspaces, history and environments

When you work with R, you create variables, functions and other objects. They appear in the "Environment" tab in RStudio and constitute what is known as a **workspace**.

worspace

When you exit R, for example when closing RStudio or switching to another project, R / RStudio will ask you whether you wish to save the current workspace and / or history. This can create two files in your project directory: `.RData` and `.Rhistory` (or `_RData` and `_Rhistory` on Windows). The `.RData` file contains the workspace, that is, all the objects (variables, functions, etc.) that you have created. The `.Rhistory` is a text file (you can open it in any text editor) that contains the history of commands that you have typed in the console.

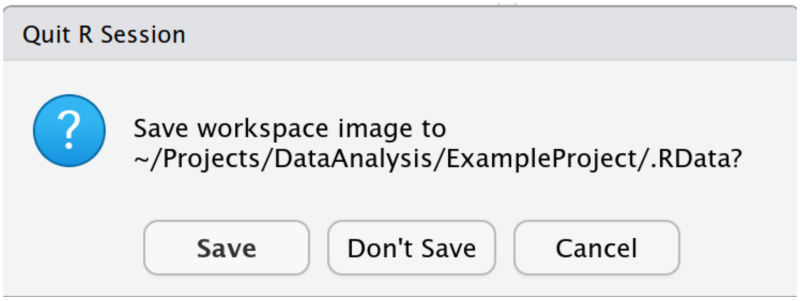


Figure 1.4: Save the workspace?

That all sounds like a good and useful thing, right? By saving the workspace, you do not have to repeat all your calculations! And by saving the history, you don't even need to type your code in a script, since R saves everything automatically, right?

Well, not so fast. Relying on that can get you into trouble.

Workspace. Saving a workspace can be useful, but it can also get you in a mess. When working with R interactively, one tends to create a lot of objects just to try out various things. Not all of them will go into your final version of the script, but the mere fact that they exist in your environment can be a problem: they can interfere with your code, take up memory and even can hide certain bugs.

Consider this example: trying out various things, you create a variable called `foo`. Later on, in a much later version of the script, you no longer create `foo`, you now have a "proper" name like `mouse_transcripts`. However, one of your script

function still uses `foo` instead of `mouse_transcripts`. You run the script and it works! But it works only as long as you have the `foo` variable in your workspace. If you were to run the script on a different computer, or give it to your colleague, it would stop working.

History. Beware! By default, R saves only the last 1000 lines of your history.

> **!** History and Workspace
>
> Do not rely on the history and workspace to save your work. Always save your scripts, and if you want to save the data, save it in a separate file.

1.5 Vectors and vectorization

1.5.1 Vectors

Variables can (and do) store a lot more than single values. One of the most basic and important data types in R is a **vector**. A vector is simply a sequence of values – just like in maths. And you know what? You have already created vectors in R. In mathematics, any scalar value can be treated as a one-dimensional vector and it is exactly like that in R: any single value is a 1-element vector, including all the variables that you have created in the previous exercise.

To create a vector with more than one value, you can use the `c()` function ("c" stands for "combine"). For integer numbers, you can use the `:` operator to create a sequence of numbers. For example:

`c()`

Creating a sequence of numbers with `:`

```
sequence <- 5:15
numbers <- c(10, 42, 33, 14, 25)
person <- c("January", "Weiner", "Hoppegarten")
```

It is also possible to combine two vectors longer than 1 into one:

Combining vectors

```r
first_v <- c(1, 2, 3)
second_v <- c(4, 5)
combined_v <- c(first_v, second_v)
print(combined_v)
```

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

Exercise 1.8 (Vectors)**.** Create a vector that combines the numeric value `1` and the string `"one"`. What happens? Can you venture a guess?

Solution

```r
combined <- c(1, "one")
combined
```

```
[1] "1"   "one"
```

In the resulting vector, the `1` is shown with quotes around it. This means that it is treated as a character string, not as a numeric value. The reason for that is that vectors can hold only one type of values. We will discuss it in more detail later.

You can access individual elements of a vector using the `[ ]` operator:

Accessing elements of a vector with `[ ]`

```r
numbers[1]
```

```
[1] 10
```

```r
person[2]
```

```
[1] "Weiner"
```

But hey, I told you that every value is a vector in R, right? And that includes the indices `1` and `2` that you have just used. So, what would happen if we used more than two values as an index? Try it:

```
numbers[1:3]
```

```
[1] 10 42 33
```

```
person[3:1]
```

```
[1] "Hoppegarten" "Weiner"      "January"
```

```
sel <- c(1, 5, 3)
numbers[sel]
```

```
[1] 10 25 33
```

As you can see, not only can you use a vector as an index, but you can also use a *variable* as an index.

> **Do not use a comma**
>
> It is tempting to select, say, first and the third element of a vector `numbers` by writing `numbers[1, 3]`. This will not work! As you will see tomorrow, this way of writing is for two-dimensional objects. You must use a vector as an index: `numbers[c(1, 3)]`.

> **Vectors and indices**
>
> In many (most?) programming languages, the first element of a vector is accessed using the index 0. For example in Python, to access the first element of an array, you need to type `array[0]`. This has something to do with how computers work. In R, the first element is always `1` – R was designed by statisticians, and in mathematics we always start counting from 1. For some reason, this seems to make some computer scientists angry.

1.5.2 Named vectors

Accessing elements of a vector using indices is all well and good, but sometimes it can be very inconvenient, especially if the vectors are very long. Or maybe you do not remember the order in which you have stored the elements of the vector – was the last name first, or second element of the `person` vector?

Vectors allow you to name their elements. We can either define the names at the very beginning, when we create the vector, or we can add them later using the `names()` function. Here is how you can do it:

Named vectors

```
person <- c(first="January", last="Weiner", city="Hoppegarten")
```

Once you have named the elements of a vector, you can access them using their names:

```
person["city"]
```

```
         city
"Hoppegarten"
```

```
person[c("first", "last")]
```

```
    first      last
"January"  "Weiner"
```

Or, we can change the names with the `names()` function:

`names()`

```
names(person) <- c("name", "given", "place")
person
```

```
        name        given        place
    "January"     "Weiner" "Hoppegarten"
```

1.5.3 Assigning values to selected elements

OK, one more thing about vectors. Above we have selected elements from a vector. It turns out, we can do more with that selections then just print it to a console:

Assigning values to selected elements

```
numbers <- c(10, 42, 33, 14, 25)
sel <- c(1, 5)
numbers[sel] <- c(100, 500)
numbers
```

```
[1] 100  42  33  14 500
```

Here is what happened: we assigned new values to the first and the fifth element of the vector `numbers`. This is a very powerful feature of R and you will be using it a lot.

Exercise 1.9 (Accessing and modifying vectors)**.**

- Create a vector with the first 10 prime numbers. Call it `primes`.
- How do you access the 3rd, 5th and 7th prime number?
- What happens when you do `primes[11]`?
- What happens when you do `primes[11] <- 31`?
- What happens when you do `primes[15] <- 47`?
- What happens when you do `primes[-1]`?
- Change the 3rd, 5th and 7th prime number to 100, 500 and 700, respectively.

Solution

```
# note that 1 is not a prime number!
primes <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
primes[c(3, 5, 7)]
```

```
[1]  5 11 17
```

```
# returns a special value, NA
# (not available)
primes[11]
```

```
[1] NA
```

```
# adds a new element to the vector
# at the end
primes[11] <- 31
primes
```

```
 [1]  2  3  5  7 11 13 17 19 23 29 31
```

```
# adds a new element to the vector
# at position 15, fills the gap with
# NAs
primes[15] <- 47
primes
```

```
 [1]  2  3  5  7 11 13 17 19 23 29 31 NA NA NA 47
```

```
# returns the vector without the first
# element
primes[-1]
```

```
 [1]  3  5  7 11 13 17 19 23 29 31 NA NA NA 47
```

```
primes[c(3, 5, 7)] <- c(100, 500, 700)
```

1.5.4 Vectorization

Vectors are very useful – but wait, there is more. What happens if we add a value to a vector? Try it:

```
numbers <- c(10, 42, 33, 14, 25)
numbers + 10
```

```
[1] 20 52 43 24 35
```

As you can see, R has added the value 10 to every single element of the vector `numbers`. The same thing happens with other operators, like `-`, `*` and `/`. Try it yourself.

This is called **vectorization** and it is one of the most powerful features of R compared to other languages. It will allow you to write very concise and, at the same time, readable code.

Vectorization

The vectorization works not only with operators like `+`, `-`, `*` and `/`, but with many functions. For example, it works with most of the mathematical functions like `sin()` or `log()`. Try it:

```
log10(numbers)
```

```
[1] 1.000000 1.623249 1.518514 1.146128 1.397940
```

```
sin(numbers)
```

```
[1] -0.5440211 -0.9165215  0.9999119  0.9906074 -0.1323518
```

However, there is a catch. What happens if you try to add two vectors when both of them with more than one element? First, let us try to add two vectors of the same length:

```
numbers1 <- c(1, 2, 3)
numbers2 <- c(4, 5, 6)
numbers1 + numbers2
```

```
[1] 5 7 9
```

As you can see, R has added the first element of the first vector to the first element of the second vector, the second element of the first vector to the second element of the second vector, and so on. Makes sense, right? Same would happen if we were to subtract, multiply or divide the vectors (or use logical operations, which you will learn on Day 3).

Imagine the two vectors one beneath the other:

```
numbers1:   c(1,    4,    5)
              +     +     +
numbers2:   c(2,    5,    6)
              ↓     ↓     ↓
result:     c(3,    9,   11)
```

R is simply adding up corresponding elements. This does not look like much now, but trust me, it will be extremely useful in the future.

However, if the vectors have *different* lengths, it is a different story altogher. Take a look:

```
numbers1 <- c(1, 2, 3)
numbers2 <- c(4, 5)
numbers1 + numbers2
```

```
Warning in numbers1 + numbers2: longer object length is not a multiple of
shorter object length
```

```
[1] 5 7 7
```

Ooops, what exactly happened here? First thing to note is that *there was no error*. There was a *warning*, but still our code executed and produced a result. But what is that result? For the first element of the result, it is clear enough: `1 + 4 = 5`. Same for the second, `2 + 5 = 7`. But what about the third? It seems that R added `3 + 4 = 7`. But why?

R noticed that it is missing an element to be added to the third element of the vector `numbers1`. So, it did what is called

recycling. It "rewound" the vector `numbers2` to the beginning and added the first element of `numbers2` to the third element of `numbers1`. However, since after the rewinding and adding one element of vector `numbers2` was left (because `numbers1` did not have any more elements), R issued a warning.

Recycling

```
numbers1:   c(1,     2,      3)
              +      +       +
numbers2:   c(4,     5)    c(4,    5)
              ↓      ↓       ↓
result:     c(3,     9,     11)
```

If the length of the first vector was a multiple of the length of the second vector, R would not have complained:

```
numbers1 <- c(1, 2, 3, 4, 5, 6)
numbers2 <- c(7, 8)
numbers1 + numbers2
```

```
[1]  8 10 10 12 12 14
```

See? No warning. R was recycling the second vector over and over again. Recycling is a dangerous business: if you are not careful, you can get results which you have not expected.

```
numbers1:   c(1,     2,      3,     4,     5,     6)
              +      +       +      +      +      +
numbers2:   c(7,     8)    c(7,     8)   c(7,     8)
              ↓      ↓       ↓      ↓      ↓      ↓
result:     c(8,    10,     10,    12,    12,    14)
```

Take it slow. This is advanced stuff, but I had to warn you already at this stage – this is one of the common sources of errors in R. Watch out for this "longer object length is not a multiple of shorter object length" warning.

Recycling advice

Here is our advice to you: either use a vector and a single element vector, or two vectors of the same length. And in the cases where, for some reason, you need to recycle, make sure that you know what you are doing. For example, check the length of both vectors.

With vectors that have only a couple of numbers it is quite easy to see what is happening, but what if you have thousands of variables? In other words, how to check the lenght of a vector? You can use the `length()`:

`length()`

```
length(numbers1)
```

```
[1] 6
```

```
length(numbers2)
```

```
[1] 2
```

Exercise 1.10 (Vectorisation)**.**

- Create a vector with several numbers and try to add, subtract, multiply and divide it by a single number. What happens?
- Say, you have three values which are the diameters of three circles: 1, 5 and 13. You would like to have a vector containing the areas of these circles. What is the simplest way of doing that?
- How do you check the length of this vector?
- One vector, `lengths`, contains the lengths of the sides of three rectangles, and the other, `widths`, contains their widths. Create a vector containing the areas of these rectangles.

Solution

```
# create a vector
numbers <- c(1, 2, 3, 4, 5)
# add, subtract, multiply and divide
numbers + 10
```

```
[1] 11 12 13 14 15
```

```
numbers - 10
```

```
[1] -9 -8 -7 -6 -5
```

```
numbers * 10
```

```
[1] 10 20 30 40 50
```

```
numbers / 10
```

```
[1] 0.1 0.2 0.3 0.4 0.5
```

```
diameters <- c(1, 5, 13)
areas <- pi * (diameters/2)^2
areas
```

```
[1]   0.7853982  19.6349541 132.7322896
```

```
length(areas)
```

```
[1] 3
```

```
lengths <- c(1, 2, 3)
widths <- c(4, 5, 6)
areas <- lengths * widths
areas
```

```
[1]  4 10 18
```

Messages, Warnings and Errors

R has three types of information to pass to you: messages, warnings and errors. Messages are just that – messages. Warnings are messages that tell you that something might be wrong and you should pay attention, but R will nonetheless do what you asked it to do. Errors stop the execution of your code, but warnings do not.
You should pay attention to warnings, but you do not have always to do something about them – some you can safely ignore. Errors, on the other hand, you should always fix.

1.5.5 The special value `NA`

```
NA + 1
```

```
numbers <- c(1, 2, NA, 4)
numbers * 3
```

```r
numbers <- c(1, 2, NA, 4)
mean(numbers)
```

```r
mean(numbers, na.rm=TRUE)
```

> **Tip 1: Useful functions**
>
> There is a whole bunch of functions that you can use to work with vectors, and here are some of them – with mostly self-explanatory names: `sum()`, `min()`, `max()`, `range()`, `sd()`, `var()`, `median()`, `quantile()`. Look them up in the help system by typing, for example, `?sum` in the console, and try them out to see how they work.

Descriptive statistics: `sum()`, `min()`, `max()` etc.

```r
imported_data <- c("10", "20", "30", "> 50", "40", "N.A.", "60 (unsure)")
# this will generate a warning
imported_data <- as.numeric(imported_data)
```

```r
imported_data
```

Special values

There are a few other special values in R that behave similarly to `NA`. `Inf` stands for infinity, you will get it when you divide a positive number by zero: `1/0`. `-Inf` is the negative infinity (when you divide a negative number by 0), and `NaN` stands for "Not a Number" – this is what you get when you try to subtracting `Inf - Inf` or dividing `0/0`. They have also other uses – for example, if a function wants to know how many rows of output you would like to see, and your answer is "all of them", you can use `Inf` as the number of rows.

`Inf`, `-Inf` and `NaN`

1.6 Putting it all together

1.6.1 Water lillies on a lake

1.
2.
3.

Exercise 1.11 (Modelling water lilies)**.** Take a piece of paper and a pen. Your task is to come up with a formula to describe the area of the lily on the n-th day. Write down the formula.
Hint: if you are stuck, try to calculate the area of the lily on the first few days.

```r
days <- 1:10
area <- 0.01 * 2^(days - 1)
area
```

```r
plot(days, area, type="b")
```

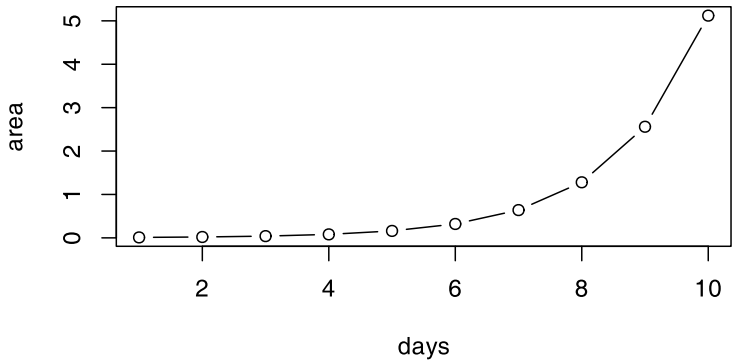
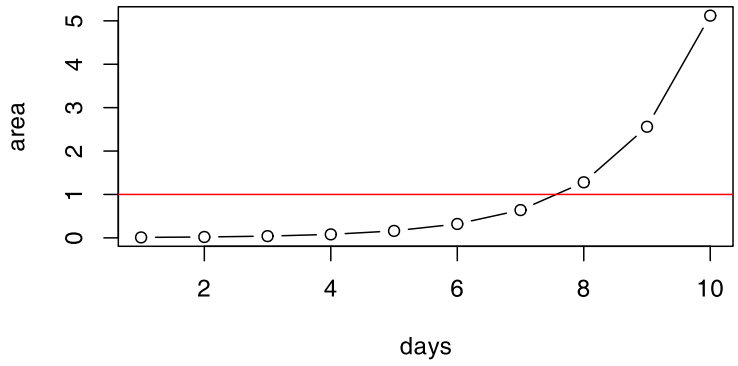


Figure 1.5

```
plot(days, area, type="b")
abline(h=1, col="red")
```



Exercise 1.12 (Plotting water lilies)**.**

- Create the same plot using the `plot()` function, but add, as a parameter, `col="blue"`. What happens?
- Now add a parameter `pch=19`. What happens?
- Use the argument `xlab` to label the x-axis with "Day". Use the argument `ylab` to label the y-axis with "Area". Use the argument `main` to give the plot a title.
- Use the argument `ylim=c(0, 1)` to change the range of the y-axis. How would you change the limit of the x axis?
- What is the formula for the area of the lily assuming that each day, the lily covers 1.75 times the area of the previous day?
- Create a new area vector (call it `area_slow`) which will be calculated with the new formula.
- Add the new vector to the plot using the function `lines()`. What does the `lines()` function do? (Hint: type `?lines` in the console).

(Solution)

1.6.2 Functions in R

```
area_lily <- function(day, fct) {
  ret <- 0.1 * fct^(day - 1)
  return(ret)
}
```

```
area <- area_lily(1:10, 2)
area
```

```
area_lily2 <- area_lily
area_lily2(1:10, 2)
```

Exercise 1.13 (Creating your own function)**.** Modify the `area_lily` function so that it takes three arguments: the day, the initial fraction and the factor. Use the new function to calculate the area of the lily on the first 10 days with the initial fraction of 0.001 and the factor of 1.5. What is the area on the 10th day?

Solution

```
area_lily <- function(day, fct, start) {
  ret <- start * fct^(day - 1)
  return(ret)
}

area_lily(1:10, 1.5, 0.001)
```

```
 [1] 0.00100000 0.00150000 0.00225000 0.00337500 0.00506250 0.00759375
 [7] 0.01139062 0.01708594 0.02562891 0.03844336
```

1.7 Coding practices

1.7.1 Computer programs as means for communication

here

1.7.2 Example

```
a<- 4
b <-c(1,10,
20, 21, 5)
r<-sqrt(sum((b-mean(b))^2)/
              a)
```

```r
# ---------------------------------------------------------
# Calculating the standard deviation of a sample
# ---------------------------------------------------------

# example values for five samples
samples <- c(1, 10, 20, 21, 5)
samples_n <- length(samples)

# calculate standard deviation of samples manually
samples_mean <- mean(samples)
samples_devs <- samples - samples_mean

# samples variance
samples_var <- sum(samples_devs^2) /
                (samples_n - 1)

samples_sd <- sqrt(samples_var)
```

```
samples_sd <- sd(samples)
```

1.7.3 Tab completion


```
samples_sd <- sd(samples)
```

1.8 Review

-
  -
  -
  -
  -
  -
-
  -
  -
  -
  -
-
  -
  -
  -
  -
  -
  -
-
  -
  -
-
  -
  -
  -
  -
  -
  -

2 Complex data structures

2.1 Goals for today

-
-

2.2 Logical vectors

2.2.1 Truth and falsehood

```
logical_vector <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
```

```
sum(logical_vector)
```

```
numbers <- 1:5
numbers[ logical_vector ]
```

TRUE, FALSE and T and F

In R, `TRUE` and `FALSE` are the only two logical constants. However, there are also two other constants, `T` and `F`, which are equivalent to `TRUE` and `FALSE`, respectively. **Do not use them**. Unlike `TRUE` and `FALSE`, they can be overwritten, which can lead to chaos and mayhem.

2.2.2 Comparison operators

-
-
-
-
-
-

```
numbers <- c(42, 3, -17, 0, -2, 1)
numbers > 0
```

```r
# prints only numbers greater than 0
numbers[ numbers > 0 ]
```

```r
# prints only numbers different from 0
numbers[ numbers != 0 ]
```

```r
numbers <- c(42, 3, NA, 0, -2, 1)
is.na(numbers)
```

```r
which(is.na(numbers))
```

```r
nas <- is.na(numbers)
nas
```

```r
# change TRUE to FALSE and FALSE to TRUE
!nas
```

```r
useful_numbers <- numbers[!nas]
useful_numbers
```

```r
patient_measurements <- c(1, 16, 7, 42, 3)
patient_gender <- c("male", "female", "female", "male", "female")
patient_measurements[ patient_gender == "male" ]
```

Exercise 2.1 (Creating logical vectors)**.** Create a vector with 50 random numbers as follows:

```r
vec <- rnorm(50)
```

How can you filter out all the numbers that are greater than 0.5?

How can you filter out all the numbers that are (i) greater than 0.5 and (ii) smaller than 1.0?
How many numbers that are greater than 0.5 are in your vector?

Solution

```
# over 0.5
over05 <- vec[ vec > 0.5 ]

# between 0.5 and 1.0
between <- over05[ over05 < 1.0 ]

# how many numbers are greater than 0.5?
sum(vec > 0.5)
```

```
[1] 17
```

2.3 Matrices

2.3.1 Creating matrices

```r
mtx <- matrix(1:12, nrow = 3, ncol = 4)
print(mtx)
```

```r
mtx <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
print(mtx)
```

```r
mtx <- matrix(0, nrow = 3, ncol = 4)
```

```r
a <- 1:3
b <- 4:6
mtx <- rbind(a, b)
print(mtx)
```

```r
mtx <- cbind(a, b)
print(mtx)
```

> **Matrices and algebra**
>
> R matrices are very powerful for linear algebra operations. If you ever learned linear algebra, you will find that R matrices can do pretty much everything you learned in class. For example, you can multiply matrices, transpose them, invert them, calculate determinants, etc. We will not cover these operations in this course.

```r
dim(mtx)
```

```r
nrow(mtx)
```

```r
ncol(mtx)
```

2.3.2 Accessing matrix elements

```r
mtx <- matrix(1:12, nrow = 3, ncol = 4)
print(mtx)
```

```r
# Accessing the element in the third row and the second column
mtx[3, 2]
```

```r
# Accessing first three numbers in the second column
mtx[1:3, 2]
```

```r
# Accessing second to fourth numbers in the first row
mtx[1, 2:4]
```

> **Rows and columns**
>
> When we talk about plotting, the first dimension, "x", is usually the horizontal one, and the second dimension, "y", is the vertical one. However, in R matrices, just like in real algebra, **the first dimension corresponds to the rows, and the second dimension corresponds to the columns**. You need to get used to it – it's the same for data frames which you will be using extensively.

```r
# Selecting the first two rows and the last two columns
# This will create a 2 x 2 matrix
mtx[1:2, 3:4]
```

```r
# Selecting the whole second row
mtx[2, ]
```

```r
# Selecting the whole third column
mtx[, 3]
```

```r
# Row 1 and three - returns a matrix
sel <- c(1, 3)
mtx[sel, ]
```

> ! Remember!
>
> - Rows first, then columns
> - If you select a single column or a single row, you will get a vector
> - If you select more than one row or column, you will get a (smaller) matrix
> - If you select more rows or columns than are present, you will get a "subscript out of bonds" error
> - Vectors and matrices **always** have only one data type (string, numerical, logical etc.)

2.3.3 Row and column names

```r
rownames(mtx) <- c("first", "second", "third")
colnames(mtx) <- LETTERS[1:4]

mtx["first", ]
```

```r
mtx[ , "A"]
```

2.3.4 Using `rep()` to generate column names

```r
abc <- LETTERS[1:3] # A, B, C
abc3 <- rep(abc, 3)
abc3
```

```
a3b3c3 <- rep(abc, each = 3)
a3b3c3
```

```
col_names <- paste0(abc3, a3b3c3)
```

```
n <- length(LETTERS)
abc3 <- rep(LETTERS, n)
a3b3c3 <- rep(LETTERS, each = n)
col_names <- paste0(abc3, a3b3c3)
length(col_names)
```

```
head(col_names)
```

```
tail(col_names)
```

Exercise 2.2 (Creating column names)**.** Repeat the procedure above, but generate column names for a matrix with more than a 1000 columns. Use the LETTERS constant, but rather then generating two-letter column names, generate three-letter column names: AAA, AAB, AAC, ..., ABA, ABC, ..., ZZZ. Store the result in a variable called `col_names`.

Exercise 2.3 (Matrices - accessing and changing elements)**.** Assume you have a 48 well-plate for a drug sensitivity analysis with viability scores.

- Create a 48-element vector "drugSensitivity_v" with random numbers between 0 and 1. Use `runif(48)` to generate these values. These reflect your viability scores.
- What does the `runif()` function do?
- Create a 6x8 matrix (6 rows, 8 columns) "drugSensitivity" from the vector.

Before starting you experiment, you decided to leave out the border wells to avoid edge effects:

- Change the values of all the border elements to NA.

The rows are treated with inhibitor 1 with increasing concentrations (control, low, medium, high). Columns 2 to

4 are treated with inhibitor 2 with increasing concentrations (control, low, high) and column 5 to 7 are treated with inhibitor 3 (same concentrations as inhibitor 2).

- Use row and column names to reflect treatments.
- Select all wells with inhibitor 3.
- Select only wells with a combination of inhibitor 1 and inhibitor 2.

(Solution)

2.4 Lists

2.4.1 Creating lists

```
lst <- list(numbers=1:3, strings=c("a", "bu"),
            matrix = matrix(1:4, nrow = 2),
            logical = c(TRUE, FALSE))
lst
```

2.4.2 Accessing elements of a list

```
lst["numbers"]
```

```
typeof(lst["numbers"])
```

```
lst[["numbers"]]
```

```
typeof(lst[["numbers"]])
```

```
lst$numbers
```

```
patient_data <- list(name = "John Doe", age = 42,
                     measurements = runif(5))
patient_data
```

```
patient_data$measurements[1]
```

```r
patient_data$measurements[1] <- 42
patient_data$measurements * 3
```

```r
names(patient_data)
```

```r
names(patient_data) <- c("patient_name", "patient_age", "patient_measurements")
names(patient_data)[3] <- "crp_measurement"
```

> **Tab completion with lists and data frames**
>
> If you type the name of your data frame variable in a script, the `$` and press the TAB key, RStudio will show you all the elements of a list (or columns of the data frame) to choose from. No need for tedious typing!

Exercise 2.4 (Lists)**.** Create a list called `misc` containing the following elements:

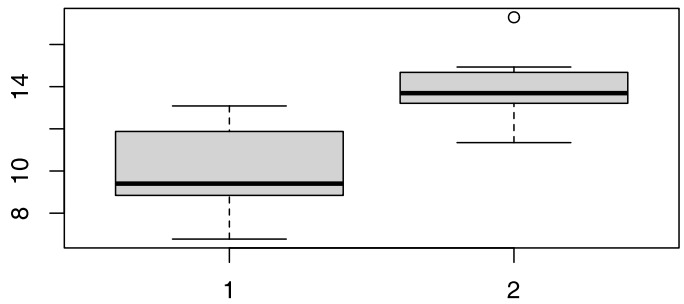
- `vector`, a vector of numbers from 1 to 5
- `matrix`, a matrix with 2 rows and 3 columns, filled with numbers from 1 to 6
- `logical`, a logical vector with three elements: `TRUE`, `FALSE`, `TRUE`.
- `person` – another list, which contains your first (`first`) and last (`last`) name

What happens when you type `misc[2:4]`?
Click on the `misc` list in the Environment tab in RStudio. What do you see? Does it make sense?

2.4.3 Lists as return values

```
group_a <- rnorm(10, mean = 10, sd = 2)
group_b <- rnorm(10, mean = 14, sd = 2)
```

```
boxplot(group_a, group_b)
```



Boxplot

Boxplots are a great way to visualize the data. The whiskers show the minimum and maximum values (excluding outliers, which are shown as separate points), the box shows the interquartile range (25th to 75th percentile), and the thick line in the middle of the box shows the median. There are better ways, which we will discuss on Day 5, but still, boxplots are pretty cool.

```
t_test <- t.test(group_a, group_b)
typeof(t_test)
```

```
t_test
```

```
t_test$p.value
```

2.4.4 Replacing, adding and removing elements of a list

```
person <- list(name = "January", age = 117, pets=c("cat", "dog"))

# change the age
person$age <- 118

# add a new element
person$city <- "Hoppegarten"

# remove pets
person$pets <- NULL
person
```

2.5 Data Frames

2.5.1 Creating data frames

```
names <- c("January", "William", "Bill")
lastn <- c("Weiner", "Shakespeare", "Gates")
age   <- c(NA, 460, 65)

people <- data.frame(names=names, last_names=lastn, age=age)
people
```

```
names(people)
```

```
colnames(people)
```

```
dim(people)
```

```
nrow(people)
```

```
ncol(people)
```

Exercise 2.5.

- Create a 5x3 matrix with random numbers. Use

`matrix` and `rnorm`.

- Turn the matrix into a data frame. Use `as.data.frame` for that.
- Add column and row names.
- Add a column. Each value in the column should be "A" (a string). Use the `rep` function for that.
- Add a column with five numbers from 0 to 1. Use the `seq` function for that. Hint: look at the help for the `seq` function (`?seq`).

(Solution)

2.5.2 Accessing and modifying columns of a data frame

```
people$names
```

```
people[1, ]
```

> **Caution**
>
> There is an issue when you try to access a single column of a data frame. We will discuss it at length in the following, but basically, this behavior is different for different *flavors* of data frames: base R data frames created with `data.frame()` return a vector, while others may return a data frame. Watch out for this!

2.5.3 Subsetting data frames with logical vectors

```
people_with_age <- people[!is.na(people$age), ]
people_with_age
```

-
-

-

2.6 Libraries in R

2.6.1 Installing and loading packages

package '—' was built under R version x.y.z

At times you will get a warning that a package was "built" under a different version of R than the one that you are running. It is not an error (just a warning), and most of the time it can be ignored. It means what it says: that the installed package was built (e.g., compiled) with a different version of R. This can sometimes lead to problems, but most of the time it does not.

> **!** Remember!
>
> Remember: you need to install a package only once with `install.package()`, but you need to load it every time you start a new R session with `library()`.

Exercise 2.6. Use `install.packages` to install the package `skimr` from CRAN. Then, load the package using `library(skimr)`. What does that package do? How can you check that? (Hint: use `??skimr`, `?skim`).

2.6.2 Data frames, tibbles & co.: different flavors of R

```
df <- data.frame(a = 1:3, b = c("a", "b", "c"))
```

```
is.data.frame(df[ , 1])
```

```
is.data.frame(df[1, ])
```

2.6.3 Data frames and tibbles

-
-

-

```
library(tidyverse)
tbl <- tibble(a = 1:3, b = c("a", "b", "c"))
tbl
```

```
is.data.frame(tbl)
```

```
is_tibble(tbl)
```

```
library(S4Vectors)
DF <- DataFrame(a = 1:3, b = c("a", "b", "c"))
DF
```

```
is.data.frame(DF)
```

```
library(tidyverse)
filter(DF, a > 1)
```

2.7 Review

-
  -
  -
  -
  -
  -
-
  -
  -
  -
  -
  -
-
  -
  -
  -
  -
-
  -
  -
  -
  -
  -
  -
-
  -

–
–

–
–
–
–
–
–
–

3 Reading and Writing Files

3.1 Aims for today

-
-
-
-

3.2 Reading data

3.2.1 Data types

-
-
-
-

```
install.packages("tidyverse")
install.packages("readxl")
```

> **!** Remember!
>
> - For reading text files (csv, tsv etc.), use the `readr` package. This package is loaded automatically when you load the `tidyverse` package: `library(tidyverse)`. Then, use the functions `read_csv`, `read_tsv` etc.
> - For reading Excel files, use the `readxl` package: `library(readxl)`. Then, use the function `read_excel`.

3.2.2 Where is my file? Relative and absolute paths

Path separators on different systems

Most computer systems separate the directories and files in a path using a slash (/). However, Windows uses a backslash (\). This is quite annoying for R users, because in R character vectors, the slash has a special meaning. To use backslashes, you need to "escape" them by putting another backslash in front of each backslash. So instead of `C:\Users\johndoe\Documents`, you need to write `C:\\Users\\johndoe\\Documents`. Alternatively, you can use the forward slash even on a Windows system, so type `C:/Users/johndoe/Documents`. We recommend the latter approach.
This is also why simply copying a path from the Windows Explorer to your R script will not work in R – because the copied text contains single backslashes.

```
library(tidyverse)
mydata <- read_csv("data.csv")
```

```
getwd()
```

Exercise 3.1 (Reading your first file)**.**

- Check your working directory using `getwd()`
- Load the tidyverse package using `library(tidyverse)`
- Go to the URL https://github.com/bihealth/RCrashcourse-book/Datasets
- Click on "iris.csv"
- Click on the "Download raw file" button on the right side of the screen
- Save the file in the directory returned by `getwd()`
- Read the file using `read_csv("iris.csv")`

```
library(tidyverse)
iris_data <- read_csv("iris.csv")
```

-
-
-

3.2.3 Reading with relative paths

Exercise 3.2 (Reading your first file from a data directory)**.**

- In the working directory, create a new directory called "Datasets"
- Move the "iris.csv" file to the "Datasets" directory
- Read the file using `read_csv("Datasets/iris.csv")`

> **!** Remember!
>
> Do not use absolute paths in your code. Always use relative paths.

3.2.4 More on relative paths

Exercise 3.3 (Reading a file from a data directory using relative paths)**.**

- In the directory that contains the working directory create a directory called "Data". That is, if your working directory is `C:/Users/johndoe/Documents/PhD/R_project`, create the directory `C:/Users/johndoe/Documents/PhD/Data`
- Move the "iris.csv" file to the new "Data" directory
- Read the file using `read_csv("../Data/iris.csv")`

3.2.5 Using RStudio to import files

Exercise 3.4 (Reading data)**.**

- Go to the URL https://github.com/bihealth/RCrashcourse-book/Datasets
- Download the following files:
  - `iris.csv`
  - `meta_data_botched.xlsx`
  - `transcriptomics_results.csv`
- Alternatively, you can download the whole repository as a ZIP file and unpack it.
- Save the files in the `Datasets/` directory in your working directory – or another location of your choice. From now on, I will assume that this is the location of your data files.
- Read the files using the appriopriate functions. Consult the table above for the correct function names, or use the RStudio data import feature. Make sure that you are using relative paths.

3.2.6 Reading Excel files

```
library(readxl)
fn <- readxl_example("deaths.xls")
print(fn)
```

```
deaths <- read_excel(fn)
```

```
head(deaths)
```

```
deaths <- read_excel(fn, skip=4)
head(deaths)
```

Exercise 3.5 (Reading data with options)**.** If you take a closer look at the file `deaths.xls`, you will notice that there is some extra text at the bottom of the file as well.

How can you omit that part when reading? *Hint: there are two ways to do that. If in doubt, look up the "examples" section of the* `readxl` *helpfile.*
(Solution)

3.3 Diagnosing and cleaning data

3.3.1 Diagnosing datasets

```r
iris_data <- read_csv("Datasets/iris.csv")
```

```r
class(iris_data[["Sepal Length"]])
```

```r
class(iris_data[["Sepal Width"]])
```

```r
class(iris_data[["Petal?Length"]])
```

```r
class(iris_data[["Petal.Width"]])
```

`class` and `typeof`

The `class` function returns the class of an object, which is a higher-level classification of the object. An object can have multiple classes. The `typeof` function returns the internal storage type of the object, which is a lower level classification. For example, both tibbles and data frames have the type `list`, but their classes are different. Another example:i if `mtx` is a matrix of numbers, `typeof(mtx)` is `double`, and `class(mtx)` is `matrix`.

```
summary(iris_data)
```

> The `summary()` functions
>
> Under the hood, there is no single `summary()` function. Instead, different classes can have different types of summaries. Whenever you produce a result, always try the summary function with it.

```
glimpse(iris_data)
```

```
str(iris_data)
```

3.3.2 Using `skim()` to diagnose datasets

```
library(skimr)
```

```
skim(iris_data)
```

Table 3.2: Data summary

```
unique(iris_data[["Species"]])
```

```
table(iris_data[["Species"]])
```

> **!** Lower and upper case
>
> For computers in general and R in particular, "lowercase" and "Uppercase" are two different things. Variables `a` and `A` are different, as are `versicolor` and `Versicolor`. This is called *case sensitivity*.

3.3.3 Checking individual values

```
sepal_length <- iris_data[["Sepal Length"]]
as.numeric(sepal_length)
```

```r
# convert to numbers
sepal_length_n <- as.numeric(sepal_length)
```

```r
# how many are NA?
sum(is.na(sepal_length_n))
```

```r
# which are NA?
which(is.na(sepal_length_n))
```

```r
# show the values
sepal_length[is.na(sepal_length_n)]
```

```r
sepal_width <- iris_data[["Sepal Width"]]

# how many are greater than 10?
sum(sepal_width > 10)
```

```r
# which are greater than 10?
which(sepal_width > 10)
```

```r
# show the values
sepal_width[sepal_width > 10]
```

Exercise 3.6 (Petal lengths)**.** Repeat the steps above for the `Petal length` column.

3.3.4 Diagnosing datasets: a checklist

Checklist for importing data

- Column names
- Data types
- Categorical variables
- Numerical variables
- Missing values

Exercise 3.7 (Botched metadata)**.** Load the file `meta_data_botched.xlsx` using the `readxl` package. Diagnose the problems.
(Solution)

3.4 Mending the data

3.4.1 Correcting column names

```
colnames(iris_data) <- c("sepal_length", "sepal_width", "petal_length",
                         "petal_width", "species")
colnames(iris_data)
```

```
library(janitor)
iris_data <- clean_names(iris_data)
colnames(iris_data)
```

3.4.2 Correcting outliers

```
# works, but don't do it
iris_data$sepal_width[42] <- 2.3
```

```
# numbers that are missing a decimal point
too_large <- iris_data$sepal_width > 10

iris_data$sepal_width[too_large] <- iris_data$sepal_width[too_large] / 10
```

```
# numbers that are missing a decimal point
sepal_width <- iris_data$sepal_width
too_large <- sepal_width > 10

sepal_width[too_large] <- sepal_width[too_large] / 10
sepal_width[too_large]
```

```
iris_data$sepal_width <- sepal_width
```

```
any(iris_data$sepal_width > 10)
```

Exercise 3.8. Can you spot what the potential problem with this approach is? Hint: what would happen if the original value was 1.10? Can you think of a better approach? Is it actually possible to make sure that our guess is correct?

3.4.3 Correcting categorical data

```
iris_data$species <- tolower(iris_data$species)
```

```
table(iris_data$species)
```

3.4.4 Incorrectly imported numeric data

Correcting values

Of course, without additional information we can only *guess* that `5,6` should be `5.6`. Maybe there were two measurements, 5 and 6? Maybe `5. 4` should be `5.04`, and not `5.4`? In a real-world scenario, you would need to consult the author(s) of the data, or check your lab book.

```
# works, but don't do it
iris_data$sepal_length[70] <- 5.6
```

```
# make a copy
sepal_length <- iris_data$sepal_length

# record the problematic places
problems <- is.na(as.numeric(sepal_length))
```

```
sepal_length[problems]
```

```
# replace the comma
sepal_length <- str_replace_all(sepal_length, ",", ".")

# what was the result?
sepal_length[problems]
```

```r
sepal_length <- str_replace_all(sepal_length, "> ", "")
```

```r
# finalize
iris_data$sepal_length <- as.numeric(sepal_length)

# check whether the column is numeric
is.numeric(iris_data$sepal_length)
```

```r
# check whether our problems are gone
iris_data$sepal_length[problems]
```

```r
# check whether there are any NA's
any(is.na(iris_data$sepal_length))
```

Exercise 3.9. Find the problems in the following vector and correct them:

```r
vec <- c(" 5", "5,6", "5.7", "5. 4", "> 5.0", "6.0")
```

3.5 Regular expressions

3.5.1 Introduction to regular expressions

```
samples <- c("ko_1_ctrl", "ko_2_ctrl", "ko_1_treat", "ko_2_treat",
             "wt_1_ctrl", "wt_2_ctrl", "wt_1_treat", "wt_2_treat")
str_detect(samples, "ctrl")
```

```
samples[str_detect(samples, "ctrl")]
```

```
samples <- c("ko_1_ctrl", "ko_2_CTRL", "ko_1_treat", "ko_2_treat",
             "wt_1_CTRL", "wt_2_ctrl", "wt_1_treat", "wt_2_treat")
# this does not work
str_detect(samples, "ctrl")
```

```
# but this does
str_detect(samples, regex("ctrl", ignore_case=TRUE))
```

```
samples <- c("ko_1_control", "ko_2_ctrl", "ko_1_trt", "ko_2_treatment",
             "wt_1_Control", "wt_2_kontrol", "wt_1_Trtmt", "wt_2_treated")
```

-
-
-
-
-
-

```
str_detect(samples, "[kcC]o?n?tro?l$")
```

```
samples[str_detect(samples, "[kcC]o?n?tro?l$")]
```

Exercise 3.10 (Quick question)**.** Look at the table above. Can you think of other patterns that would match this regular expression? Would `kotrl` work? What about `cotro`?

3.5.2 How regexp works

```
strings <- c("a", "ab", "ac", "ad", "bc", "abc", "bac", "cab")
```

```
strings[str_detect(strings, "a")]
```

```
strings[str_detect(strings, "^a")]
```

```
strings[str_detect(strings, "^a[bc]")]
```

3.5.3 Usage of regular expressions in R

-
-

```
gender <- c("m", "f", "m", "w", "frau",
            "female", "other", "male",
            "männlich", "x", "weiblich")
```

```
gender <- str_replace(gender, "^m.*", "male")
gender <- str_replace(gender, "^[fw].*", "female")
gender <- str_replace(gender, "^[^mfw].*", "other")
```

Exercise 3.11 (Quick question)**.** What would happen if we ommited the `^` at the beginning of the strings above? For example, if we used `[^mfw].*` instead of `^[^mfw].*`? Think first, then try it out.

```
vec <- c("5.6", "5.7", "5.8")
str_replace_all(vec, ".", ",")
```

```
vec <- c("5.6", "5.7", "5.8")
str_replace_all(vec, "\\.", ",")
```

Exercise 3.12.

- Use `str_replace_all()` to make the following uniform: `c("male", "Male ", "M", "F", "female", " Female")`
- Using `str_replace_all()` and `toupper()`, clean up the gene names such that they conform to the HGNC (all capital letters, no spaces, no dashes): `c("ankrd22", "ifng", "Nf-kb", " Cxcl 5", "CCL 6.", "ANK.r.d. 12")`
- What regular expression matches all of the ankyrin repeat genes (but not other genes) in the following vector: `c("ANKRD22", "ank.rep.d. 12", "ANKRD-33", "ankrd23", "ANKEN", "MAPK", "ifng-1", "ANKA-REP-6")`? Ankyrin repeat domain genes are the first 4 in the vector.

3.5.4 Correcting columns in `iris_data` with regular expressions

```
petal_length <- iris_data$petal_length
problems <- is.na(as.numeric(petal_length))
```

```
which(problems)
```

```
petal_length[problems]
```

```
petal_length <- str_replace_all(petal_length, "[^0-9.]", "")
```

```
# check the problems
petal_length[problems]
```

```
# convert to numbers
petal_length <- as.numeric(petal_length)

# check for remaining NA's
any(is.na(petal_length))
```

```
# assign
iris_data$petal_length <- as.numeric(petal_length)
```

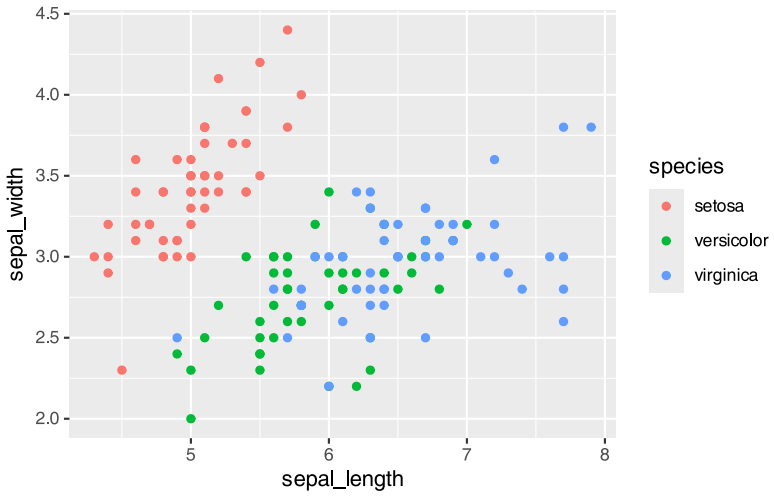
```
skim(iris_data)
```

Table 3.10: Data summary

Exercise 3.13 (Correcting metadata)**.** In Exercise 3.7, you have diagnosed the file `meta_data_botched.xlsx`. Now go ahead and correct the problems you have found.

3.5.5 Just a quick look

```
library(ggplot2)
ggplot(iris_data, aes(x=sepal_length, y=sepal_width, color=species)) +
  geom_point()
```



3.6 Writing data

3.6.1 Keep your data organized

-
-
-
-
-

3.6.2 Functions to use for writing data

Exercise 3.14 (Writing data)**.**

- In your project directory, create the directory "Data_clean" (if you want, you can use the R function `dir.create()` for that).
- Write the cleaned `iris_data` to a new file in the "Data_clean" directory. Use the `write_csv` function from the `readr` package. Use a meaningful name, version it and use a date.

3.7 Principles of data management

3.7.1 Keeping originals

3.7.2 Use R for data editing

-
-

```r
# There seems to be a missing decimal dot in some of the sepal widths
# Changing it to 1/10th of the value

sepal_width <- iris_data[["Sepal Width"]]
sel <- which(sepal_width > 10)
sepal_width[sel] <- sepal_width[sel] / 10

iris_data[["Sepal Width"]] <- sepal_width
```

3.7.3 Working with Excel and other spreadsheets

Avoid working with Excel

Excel can automatically replace strings with dates, like changing `MARCH9` into `9th of March`. However, `MARCH9` is a valid gene name. In fact, it turns out that a substantial proportion (about a third!) of all Excel files containing gene names and published as supplementary materials online contain gene names transformed to dates. Not only that, but even though that this has been discovered many years ago and even some genes were officialy renamed *because of Excel*, this is still a problem. And Excel is now able to recognize dates in many languages, exacerbating the problem (Abeysooriya et al. 2021).

-
-
-
-

-
-
-
-
- here are the instructions
-

3.8 Review

-
  -
  -
  -
-
  -

- –
-
  - –
  - –
  - –
  - –
  - –
-
  - –
  - –
  - –
  - –
-
  - –
  - –
  - –
  - –
-
  - –

4 Manipulating data frames

4.1 Aims for today

-
-
-
-

4.2 Selecting columns

4.2.1 Selecting columns using square brackets

```
df <- data.frame(ara=1:5, bera=6:10, cora=11:15, dora=16:20)
df$ara # same as df[["ara"]]
```

```r
# select columns 1 to 2
df2 <- df[ , 1:2]

# select anything but column 2
df2 <- df[ , -2]

# select all columns in reverse order
df2 <- df[ , ncol(df):1]

# select columns ara and cora
df2 <- df[ , c("ara", "cora")]

# select all columns ara and cora, but in reverse order
df2 <- df[ , c("cora", "ara")]
```

4.2.2 Selecting columns using tidyverse

```r
library(tidyverse)
# select columns ara and cora
df2 <- select(df, ara, cora)

# select columns ara to cora
df2 <- select(df, ara:cora)

# select anything but column bera
df2 <- select(df, -bera)
```

> **!** Remember!
>
> Tidy evaluation only works with tidyverse functions!

```
df2 <- select(df, ends_with("ora"))
colnames(df2)
```

> To quote or not to quote?
>
> The fact that you don't use quotes around column names in tidyverse is confusing for many. As a rule of thumb, at

this stage you should use quotes around column names, unless:

- you are using `$` operator (e.g. `df$ara`)
- you are using `select()`, `mutate()` or `filter()` from tidyverse
- you are using another tidyverse function that uses tidy evaluation (look up the help page if unsure)

Once you start to program your own functions, the tidy evaluation will be an additional source of confusion, but burn that bridge when you get to it.

4.2.3 Renaming columns

```
df2 <- select(df, Alpha=ara, Gamma=cora)
colnames(df2)
```

```
df2 <- rename(df, Alpha=ara, Gamma=cora)
colnames(df2)
```

Exercise 4.1.

- Read the file 'Datasets/transcriptomics_results.csv'
- What columns are in the file?
- Select only the columns 'GeneName', 'Description', 'logFC.F.D1' and 'qval.F.D1'
- Rename the columns to 'Gene', 'Description', 'LFC' and 'FDR'

(Solution)

4.3 Sorting and ordering

4.3.1 Sorting and ordering vectors

```
vec <- round(rnorm(10), 2)
vec
```

```
sort(vec)
```

```
sort(vec, decreasing=TRUE)
```

```
ord <- order(vec)
ord
```

```
vec[ord]
```

Exercise 4.2 (Reverse sorting)**.** How can you get the reverse order vector for vec?

4.3.2 Sorting character vectors

```r
chvec <- c("b", "a", "Zorro", "10", "Anton", "A", "2", "100", "zxy")
sort(chvec)
```

```r
chvec <- c("ID1", "ID2", "ID3", "ID4", "ID10", "ID20", "ID100")
sort(chvec)
```

```r
# extract numbers
chvec_n <- str_replace_all(chvec, "ID", "")

# convert to numeric
chvec_n <- as.numeric(chvec_n)

# get the order
```

```
ord <- order(chvec_n)

# sort the original vector
chvec[ord]
```

Exercise 4.3 (Sorting by last name)**.** Here is a vector of names. Can you think of how to sort it by last name?

```
persons <- c("Henry Fonda", "Bob Marley", "Robert F. Kennedy",
  "Bob Dylan", "Alan Rickman")
```

(Solution)

4.3.3 Sorting data frames with `order()`

4.1

```
library(tidyverse)
tr_res <- read_csv("Datasets/transcriptomics_results.csv")
tr_res <- select(tr_res,
                 Gene=GeneName, Description,
                 logFC=logFC.F.D1, FDR=qval.F.D1)
```

```
ord <- order(tr_res$logFC, decreasing=TRUE)
head(tr_res[ord, ])
```

Exercise 4.4 (Sorting data frames with order()).

- Sort the data frame `tr_res` by increasing `FDR`
- Sort the data frame alphabetically by `Gene`

4.3.4 Sorting data frames with tidyverse

```
tr_res <- arrange(tr_res, desc(logFC))
head(tr_res)
```

```
tr_res <- arrange(tr_res, logFC, FDR)
head(tr_res)
```

```
tr_res <- arrange(tr_res, logFC * FDR)
```

Exercise 4.5 (Sorting data frames with arrange()). The `abs()` function returns the absolute value of a number, for example `abs(c(-3, 7))` returns `3, 7`. Use the `arrange()` function to sort the data frame `tr_res` by the decreasing absolute value of `logFC`.

4.4 Modifying and filtering data frames in tidyverse

4.4.1 Modifying with `mutate()`

```r
tr_res <- mutate(tr_res, logFC_abs = abs(logFC))
```

```r
tr_res$logFC_abs <- abs(tr_res$logFC)
```

4.4.2 Using `ifelse()` with `mutate()`

```r
# default value
tr_res$direction <- "up"

# create logical vector
negative <- tr_res$logFC < 0

# change "up" to "down" where logFC < 0
tr_res$direction[negative] <- "down"
```

```
tr_res <- mutate(tr_res,
                 direction = ifelse(logFC < 0, "down", "up"))
```

4.4.3 Filtering with logical vectors and `filter()`

```
small_fdr <- tr_res$FDR < 0.05
significant <- tr_res[small_fdr, ]
```

```
significant <- filter(tr_res, FDR < 0.05)
```

```
interferon <- filter(tr_res,
                     str_detect(Description, "interferon"))
```

```
significant <- filter(tr_res, FDR < 0.05, abs(logFC) > 1)
```

```
large_fc <- abs(tr_res$logFC) > 1
significant <- tr_res[small_fdr & large_fc, ]
```

```
significant <- filter(tr_res, FDR < 0.05 & abs(logFC) > 1)
```

```
vec1 <- c(TRUE, TRUE, FALSE, FALSE)
vec2 <- c(TRUE, FALSE, TRUE, FALSE)
vec1 & vec2
```

Exercise 4.6 (Logical vectors)**.**

- Create a logical vector that selects genes that are both significant (FDR < 0.05) and contain the string "interferon" in the Description. How many are there? How many genes are significant and do not contain "interferon" in the Description column?
- How many of the genes containing "interferon" in the Description are are guanylate binding proteins? You can recognize the GBPs by their Gene symbol – it starts with "GBP". (Solution) gical

4.4.4 Using the %in% operator

```
vec1 <- c("a", "b", "c")
vec2 <- c("a", "b", "x", "y", "z")
vec1 %in% vec2
```

```
genes <- c("GBP1", "GBP2", "GBP3", "GBP4", "GBP5", "ANKRD22")
filter(tr_res, Gene %in% genes)
```

Exercise 4.7. Which of these genes are significant? Use the `&` operator to combine the result from `%in%` with the result of `FDR < 0.05`.

4.5 Combining data sets

4.5.1 Binding data frames with `rbind()` and removing duplicates

```
interferon <- filter(tr_res,
                     str_detect(Description, "interferon"))
gbp <- filter(tr_res,
              str_detect(Gene, "GBP"))
results <- rbind(interferon, gbp)
```

```
a <- data.frame(a=1:5, b=6:10)
b <- data.frame(a=11:15, c=16:20)
rbind(a, b)
```

```
intersect(interferon$Gene, gbp$Gene)
```

```
# count the number of the duplicated results
sum(duplicated(results$Gene))
```

```
# number of rows before removing dups
nrow(results)
```

```
# remove duplicates
results <- filter(results, !duplicated(Gene))

# how many are left?
nrow(results)
```

Exercise 4.8. You might have been expecting that there are only three duplicates; after all, only four genes were in common between `interferon` and `gbp`- So where do these other duplicates come from? What are they?

4.5.2 Binding data frames with `cbind()`

```
df1 <- data.frame(a = 1:4, b=11:14)
df2 <- data.frame(c=21:24)
cbind(df1, df2)
```

4.5.3 Merging data sets

```
ids_x <- paste0("ID", sample(1:8, 6))
df_x <- data.frame(ID=ids_x, val_x=rnorm(6))
df_x
```

```
ids_y <- paste0("ID", sample(1:8, 6))
df_y <- data.frame(ID=ids_y, val_y=rnorm(6, 10, 1))
df_y
```

```r
intersect(ids_x, ids_y)
```

```r
# inner join by ID
joined <- merge(df_x, df_y, by="ID")
joined
```

```r
# full join by ID
joined <- merge(df_x, df_y, by="ID", all=TRUE)
joined
```

```r
# left join by ID
joined <- merge(df_x, df_y, by="ID", all.x = TRUE)
joined
```

```r
# right join by ID
joined <- merge(df_x, df_y, by="ID", all.y = TRUE)
joined
```

```r
df_x <- data.frame(ID=ids_x, val_x=rnorm(6))
df_y <- data.frame(PatientID=ids_y, val_y=rnorm(6, 10, 1))

# inner join by ID
joined <- merge(df_x, df_y, by.x="ID", by.y="PatientID")
```

4.5.4 Complex merges

```r
meta_data <- data.frame(ID=paste0("ID", 1:4),
                        age=sample(1:3, 4, replace=TRUE),
                        group=rep(c("control", "treated"), each=2))
meta_data
```

```r
crp <- data.frame(ID=rep(paste0("ID", 1:4), each=2),
                          time=rep(c("day1", "day2"), 4),
                          CRP=rnorm(8))
crp
```

```
merged_data <- merge(meta_data, crp, by="ID")
merged_data
```

```
albumin <- data.frame(ID=rep(paste0("ID", 1:4), each=2),
                      time=rep(c("day1", "day2"), 4),
                      ALB=rnorm(8))

# incorrect code!!!
crp_alb <- merge(crp, albumin, by="ID")
crp_alb
```

```
crp_alb <- merge(crp, albumin, by=c("ID", "time"))
crp_alb
```

4.5.5 Bringing it all together

Exercise 4.9 (Merging large data sets)**.** The files `expression_data_vaccination_example.xlsx` and `labresults_full.csv` contain data from the same study. The first contains some meta-data for samples for which gene expression has been measured, and the latter contains the results of laboratory tests for many patients in the trial. The goal here is to connect the RNA expression data with the laboratory values, so that one can analyze the relationship between the expression of certain genes and, say, the observed levels of inflammation.

1. Read CSV file and the first sheet ("targets") from the XLSX file.
2. Which columns contain the ID the subjects? Are there any subjects in common? (Hint: use the `intersect()` function)
3. What other columns are common between the two data frames? Is the column with the ID of the subject sufficient to identify each row?
4. We are interested only in the following information: SUBJ, ARM (group), time point, sex, age, test name

and the actual measurement. Are the measurements numeric? Remember, you can use expressions like `[ , c("ARM", "sex") ]` or `select(df, ARM, sex)` to select the desired columns from a data set.

5. Use the subjects and / or other information to merge the two data frames however you see fit. Note that there are multiple time points per subject and multiple measurements per subject and time point.

See Solution

4.6 Pipes in R

4.6.1 Too many parentheses

```
vec <- c(1, NA, 3, 10)
which(is.na(as.numeric(vec)))
```

```
library(tidyverse)
library(janitor)
iris_data <- read_csv("Datasets/iris.csv")
iris_data <- clean_names(iris_data)
iris_data$species <- tolower(iris_data$species)
iris_data$petal_length <- str_replace_all(iris_data$petal_length ,"[^0-9.]", "")
iris_data$petal_length <- as.numeric(iris_data$petal_length)
iris_data$sepal_length <- str_replace_all(iris_data$sepal_length, ",", ".")
```

```
iris_data$sepal_length <- str_replace_all(iris_data$sepal_length ,"> ", "")
iris_data$sepal_length <- as.numeric(iris_data$sepal_length)
```

```
iris_data <-
  mutate(mutate(mutate(clean_names(read_csv("Datasets/iris.csv")),
  petal_length = as.numeric(
  str_replace_all(petal_length, "[^0-9.]", ""))),
  sepal_length = as.numeric(str_replace_all(
  str_replace_all(sepal_length, ",", "."), "> ", ""))),
  species = tolower(species))
```

4.6.2 Pipes

```
vec <- c(1, NA, 3, 10)
as.numeric(vec)
```

```
vec |> as.numeric()
```

```
which(is.na(as.numeric(vec)))
```

```
vec |> as.numeric() |> is.na() |> which()
```

```
iris_data <- read_csv("Datasets/iris.csv") |>
        clean_names() |>
        mutate(species = tolower(species)) |>
        mutate(petal_length = str_replace_all(petal_length ,"[^0-9.]", "")) |>
        mutate(petal_length = as.numeric(petal_length)) |>
        mutate(sepal_length = str_replace_all(sepal_length, ",", ".")) |>
        mutate(sepal_length = str_replace_all(sepal_length ,"> ", "")) |>
        mutate(sepal_length = as.numeric(sepal_length))
```

```
iris_data <- read_csv("Datasets/iris.csv") |>
        clean_names() |>
        mutate(species = tolower(species))

iris_data$petal_length <- iris_data$petal_length |>
        str_replace_all("[^0-9.]", "") |>
        as.numeric()

iris_data$sepal_length <- iris_data$sepal_length |>
        str_replace_all(",", ".") |>
        str_replace_all("> ", "") |>
        as.numeric()
```

here

Exercise 4.10 (Botched meta data)**.** Go back to the code that you have used yesterday for cleaning up the dataset from file `meta_data_botched.xlsx`. Use pipes to make it more readable.

4.7 Review

-
  -
  -
  -
  -
-
  -
  -
  -
  -
  -
  -
  -
  -
  -
-
  -
  -
  -
-
  -
  -
-
  -
  -
  -

5 R markdown, basic statistics and visualizations

5.1 Aims for today

-
-
-
-

5.2 R markdown

5.2.1 What is R markdown?

github repository

here

-
-
-
-
-
-
-

-

R markdown vs Quarto

- R markdown is older and more widely supported
- Quarto is newer, slightly more complex, with additional features and generally better looking

Documents created in Quarto can largely be processed with R markdown and vice versa, only some visual bells & whistles might be lacking.
Both Quarto and R markdown are available if you have installed RStudio. Standalone R installations without RStudio may require additional packages (e.g. `rmarkdown` for R markdown) or programs (`quarto` for Quarto).

Exercise 5.1 (Create an R markdown document)**.** Go to the "File" menu in Rstudio and choose "New File" -> "R markdown". Enter a title and click on "OK". Save the file in your project directory and take a look at it. Rstudio has created a simple R markdown file for you so that you might get an idea of how it works.
Click on the "knit" icon in the toolbar to render the document. What do you see? How does it relate to the content of the document? Try changing a few words in the document and click on "knit" again. What happens?

5.2.2 Markdown basics

take a look!

link to website

-
-
  - –
  - –
-

markdown guide
Quarto markdown documentation

Mathematical formulas

Using a special syntax, it is possible to include virtually any mathematical formula in R, both inline variant (like $\sigma = \sqrt{\frac{(x_i-\bar{x})^2}{n}}$) or as a stand-alone block:

$$\sigma = \sqrt{\frac{(x_i - \bar{x})^2}{n}}$$

When you convert the R markdown document to Word, you will even be able to edit the formulas natively in Word. There are several formulas in this book, if you are interested, look up the book quarto sources on github or check this guide.

Exercise 5.2 (Formatting markdown)**.**

- Try some formatting commands in markdown. For example, try to make a word **bold** or *italic*. Try to create a list. Insert a link to a website.
- This one is a bit harder, because I did not show it to you (on purpose). First download any kind of image from the internet and save it in your project directory. Then, using whatever means necessary (Google, the markdown guide, the Rstudio help), try to figure out how to insert an image in your document.

5.2.3 R markdown header

```
---
title: "Untitled"
author: "JW"
date: "`r Sys.Date()`"
output: html_document
---
```

```
Sys.Date()
```

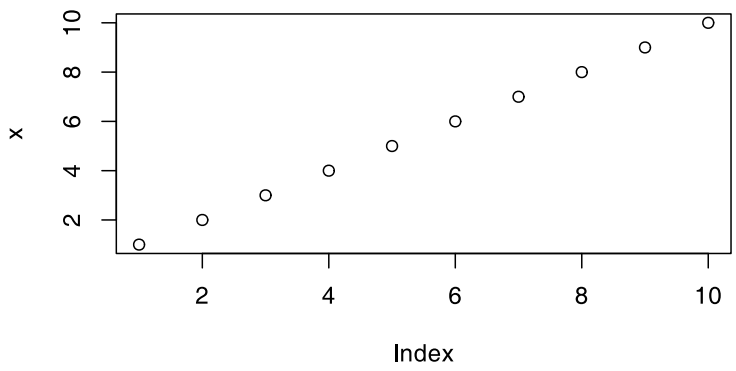
Exercise 5.3 (Choosing output format)**.** When editing your R markdown document, click on the little arrow next to the "Knit" icon in RStudio. Choose the PDF format. Observe what happens to the header of your document.

5.2.4 R code chunks

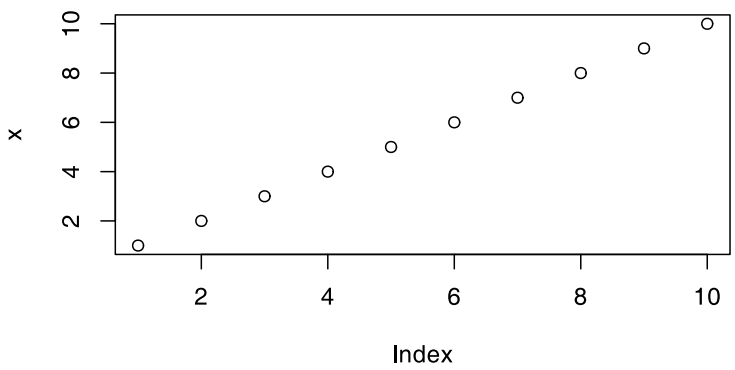
````
```{r}
x <- 1:10
print(x)
plot(x)
```
````

```r
x <- 1:10
print(x)
```

```r
plot(x)
```



```{r echo=FALSE}
x <- 1:10
print(x)
plot(x)
```



Code chunks run in one environment

Code chunks share their environment. That is, if you define a variable in one chunk, you will be able to use it in the subsequent chunks.
Think of all the chunks as consecutive fragments of your script file!

```
For example, when I write that the $\pi$ constant is
equal to `r pi`, I am using an inline code chunk:
```

Exercise 5.4 (Put your code in your markdown)**.**

- Go back to Day 1, the water lillies example in Section 1.6.1. Copy the code to the R markdown, including the plot and Exercise 1.11.
- Write a brief summary including inline code chunks that show the number of days of the simulation used.

5.3 Visualizations with R

5.3.1 Basic principles of visualization

Visualization principles

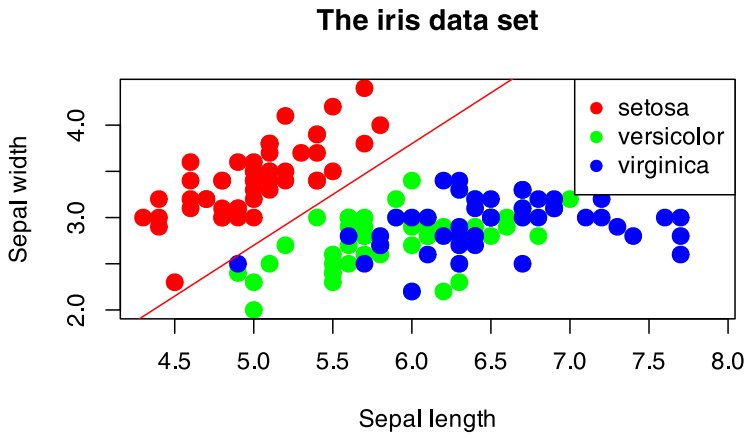
- **Message**: think what the message of your plot is
- **Design**: make a sketch on how to best communicate the message
- **Inspiration**: look at other people's plots for inspiration
- **Simplicity**: less is more
- **Accessibility**: make your fonts large and colors color-blind friendly

5.3.2 Base R vs `ggplot2`

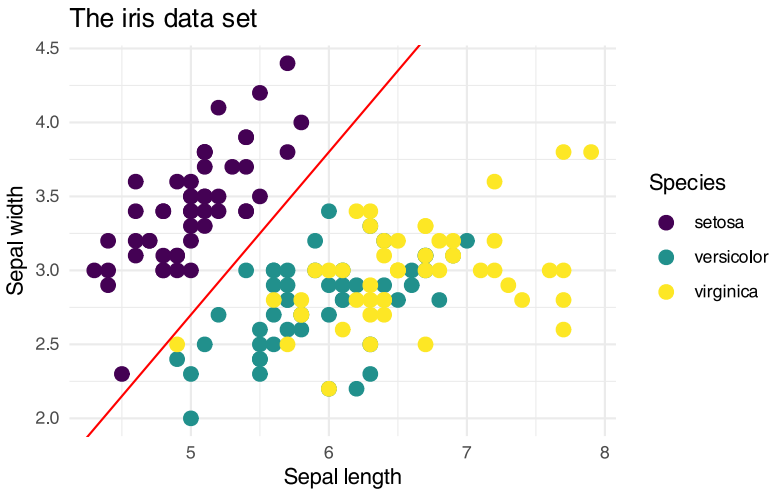
1.6.1

3.5.5

```r
colors_map <- c(setosa="red", versicolor="green", virginica="blue")
colors <- colors_map[iris$Species]
plot(iris$Sepal.Length, iris$Sepal.Width,
     col=colors, pch=19, xlab="Sepal length",
     cex=1.5,
     ylab="Sepal width", main="The iris data set")
abline(-2.8, 1.1, col="red")
legend("topright", legend=unique(iris$Species),
       col=colors_map[unique(iris$Species)], pch=19)
```



```
library(ggplot2)
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point(size=3) +
  geom_abline(intercept = -2.8, slope = 1.1, color = "red") +
  scale_color_viridis_d() +
  labs(x = "Sepal length", y = "Sepal width",
       title = "The iris data set") +
  theme_minimal()
```



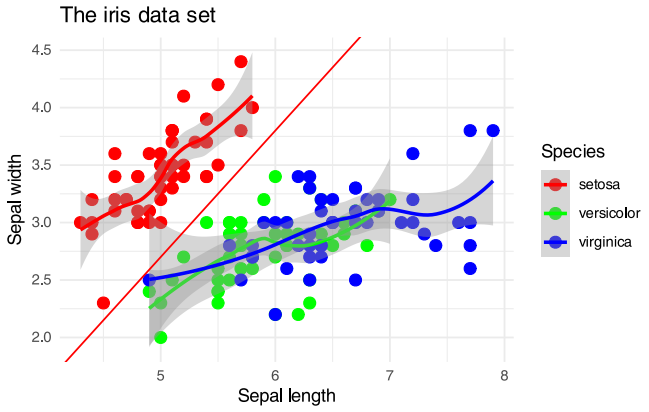
Exercise 5.5 (Ggplot2)**.** Take the code for the `ggplot2` plot above and make the following modifications:

- Try adding a `geom_smooth()` layer
- Remove `scale_color_viridis_d()` layer. What happens?
- Add the following layer: `scale_color_manual(values=colors_map)`. What happens?
- Change the color of all points to red (use the `color` parameter in `geom_point()`)
- If you wanted to show different shapes for different species rather than different colors, where would you change the plot? Hint: where is the species mentioned in the code?

Solution

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) +
  geom_point(size=3) +
  scale_color_manual(values=colors_map) +
  # LOESS smoothing, separate for each Species
  geom_smooth() +
  geom_abline(intercept = -2.8, slope = 1.1, color = "red") +
  labs(x = "Sepal length", y = "Sepal width",
       title = "The iris data set") +
  theme_minimal()
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



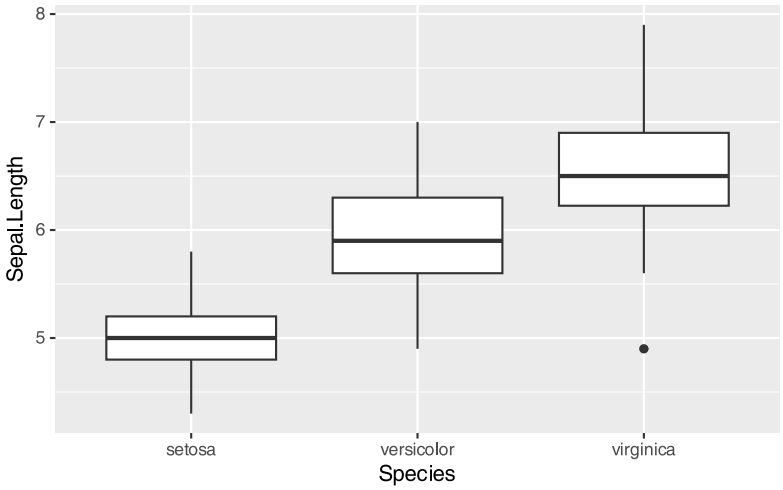
5.3.3 Esthetics and information channels

5.3.4 Boxplots and violin plots

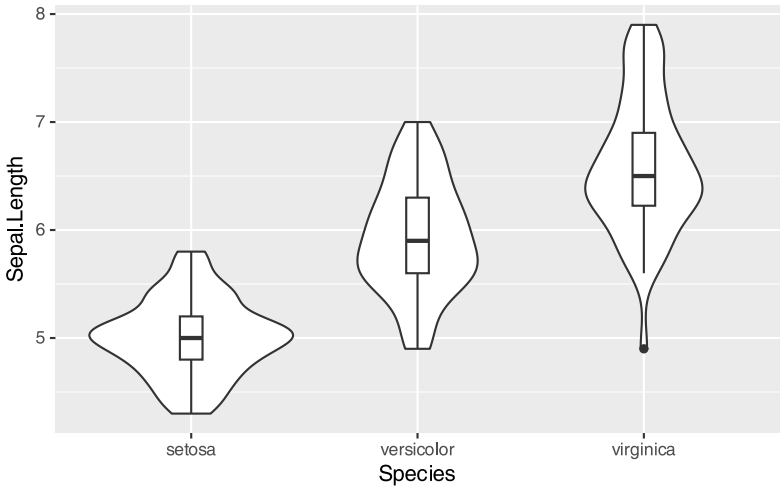
R Graphics Cookbook

2.4.3

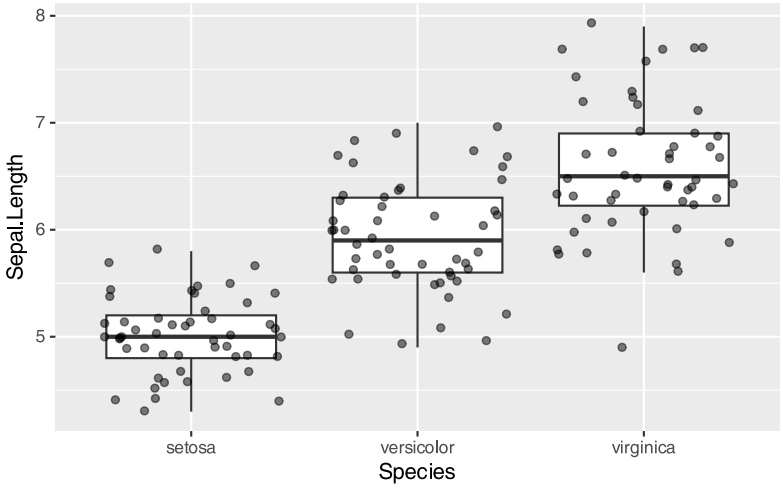
```
library(ggplot2)
ggplot(iris, aes(x=Species, y=Sepal.Length)) +
  geom_boxplot()
```



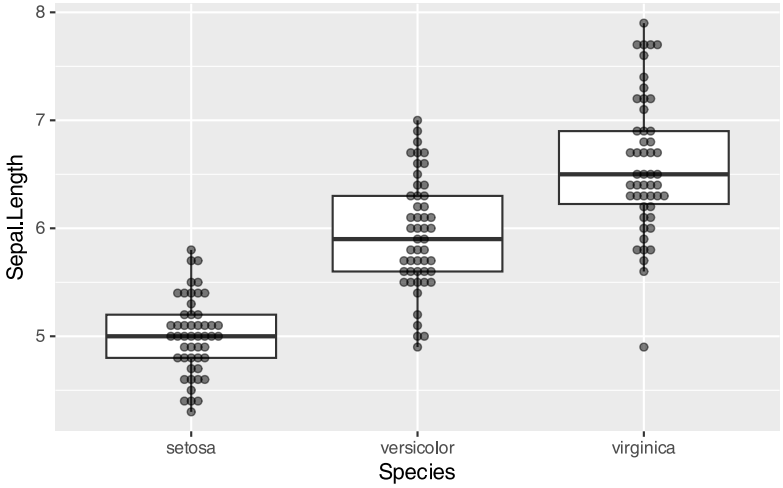

```
ggplot(iris, aes(x=Species, y=Sepal.Length)) +
  geom_violin() +
  geom_boxplot(width=0.1)
```




```
ggplot(iris, aes(x=Species, y=Sepal.Length)) +
  geom_boxplot(outlier.shape=NA) +
  geom_jitter(alpha=.5)
```



```
library(ggbeeswarm)
ggplot(iris, aes(x=Species, y=Sepal.Length)) +
  geom_boxplot(outlier.shape=NA) +
  geom_beeswarm(alpha=.5)
```

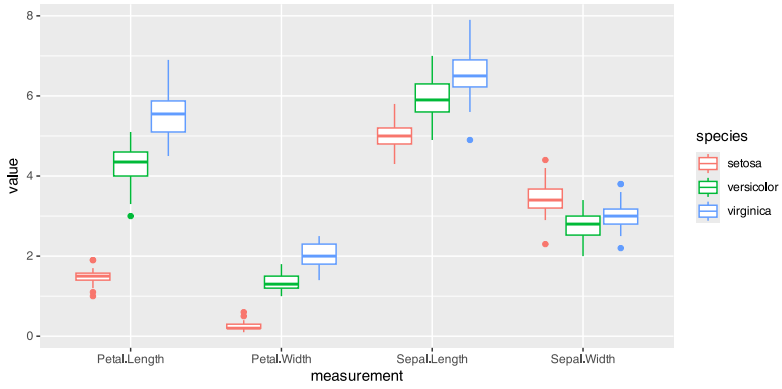


Appendix “Wide vs long data”

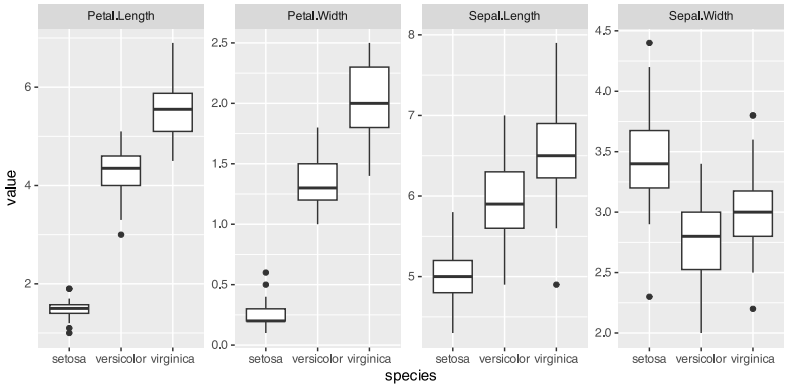
```
library(tidyverse)
df <- data.frame(species = rep(iris$Species, 4),
  measurement = rep(c("Sepal.Length", "Sepal.Width",
                      "Petal.Length", "Petal.Width"), each=150),
```

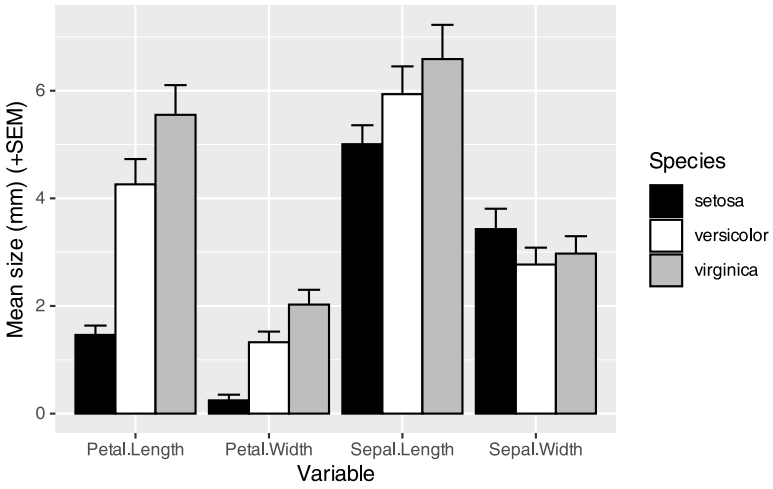
```
  value = c(iris$Sepal.Length, iris$Sepal.Width,
            iris$Petal.Length, iris$Petal.Width)
)
dim(df)
```

```
ggplot(df, aes(x=measurement, y=value, color=species)) +
  geom_boxplot()
```



```
ggplot(df, aes(x=species, y=value)) +
  geom_boxplot() +
  facet_wrap("measurement", ncol=4, scales="free_y")
```





this section in the Appendix

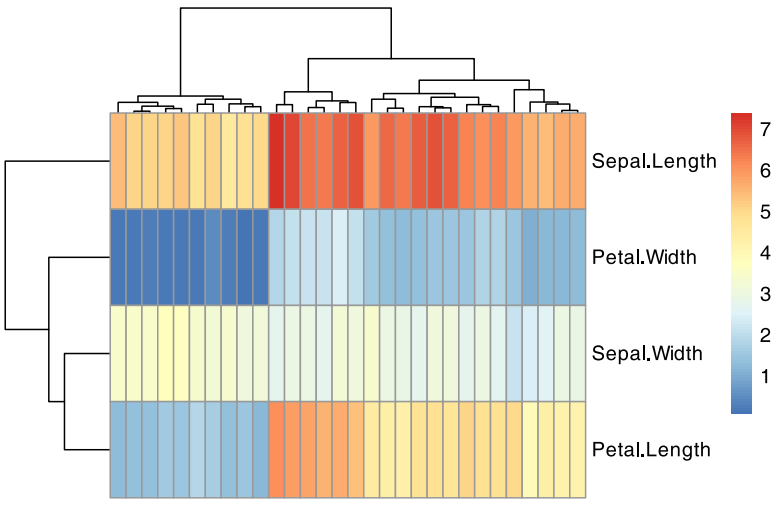
Show actual data

If possible, always strive to show the actual data points (with `geom_beeswarm()` or `geom_jitter()`) in addition to the summary statistics. If not, at least show the distribution (with `geom_violin()`). Never use a bar plot.

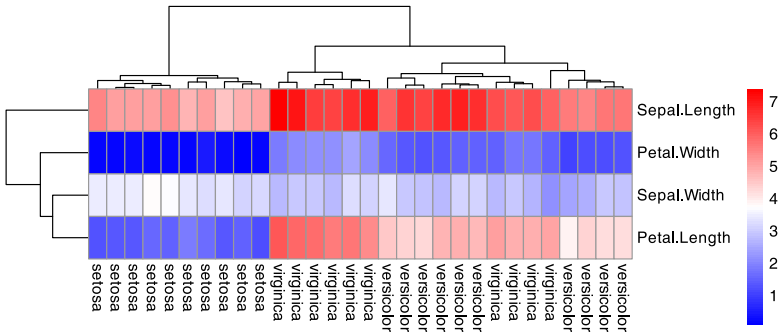
5.3.5 Heatmaps

```
library(tidyverse)

iris_sel <- iris |>
            group_by(Species) |>
            slice_sample(n=10) |>
            ungroup()
table(iris_sel$Species)
```

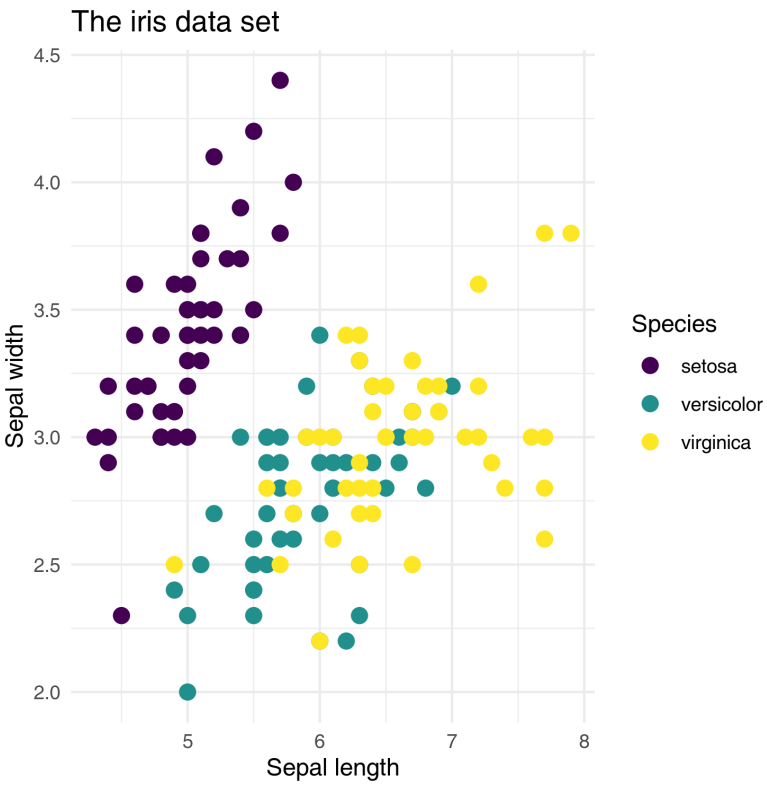
```
library(pheatmap)
iris_mtx <- t(as.matrix(iris_sel[, 1:4]))
pheatmap(iris_mtx)
```



```
iris_species <- data.frame(species=iris_sel$Species)
foo <- pheatmap(iris_mtx, labels_col=iris_sel$Species,
  color = colorRampPalette(c("blue", "white", "red"))(100)
)
```



5.3.6 Output formats


The iris data set
Sepal width
Sepal length
4.5
4.0
3.5
3.0
2.5
2.0
5
6
7
8
Species
setosa
versicolor
virginica

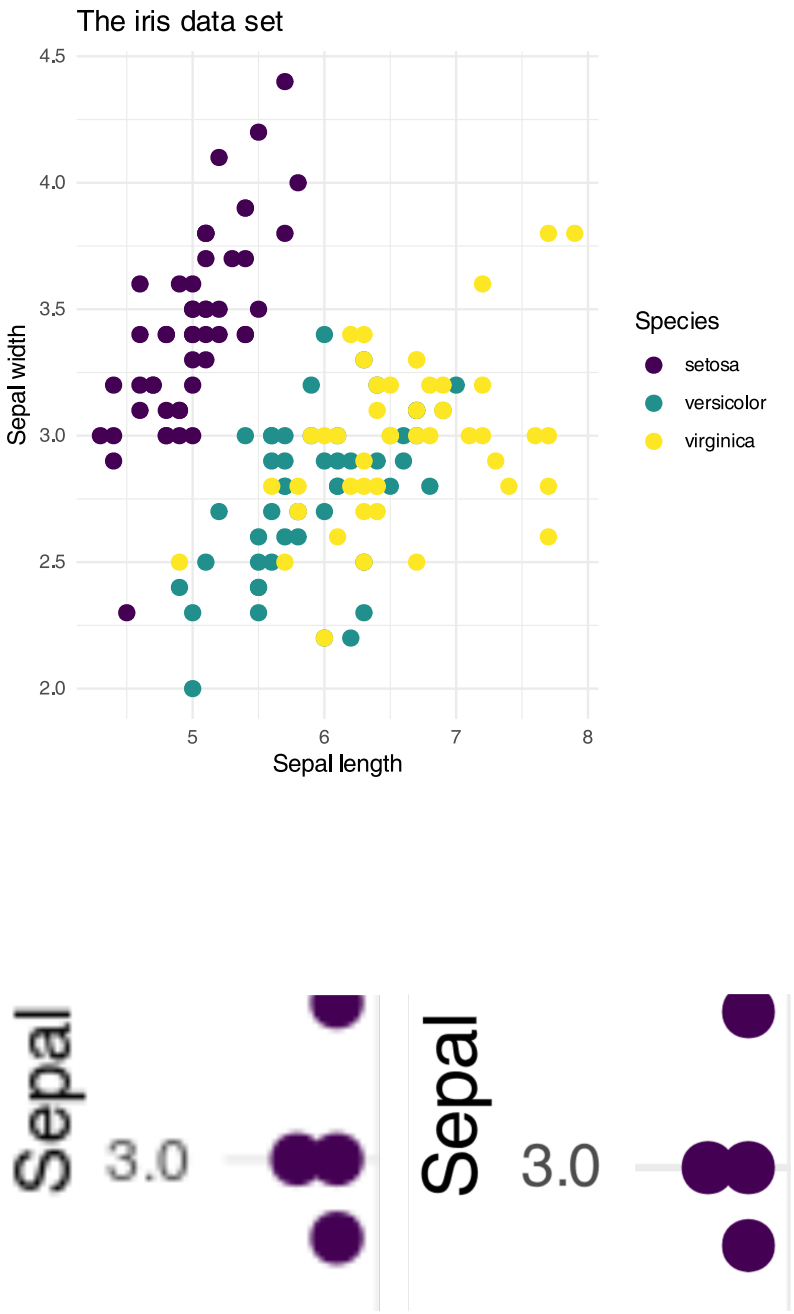
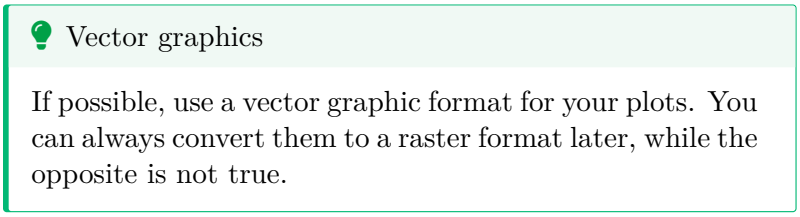


Figure 5.1: Raster vs vector graphics

Inkscape


Vector graphics
If possible, use a vector graphic format for your plots. You can always convert them to a raster format later, while the opposite is not true.

````
```{r include=FALSE}
knitr::opts_chunk$set(dev="svg")
```
````

````
```{r dev="svg"}
plot(1:10)
```
````

> **Word and R plots**
>
> When you create a Word document with Rmarkdown, **never** copy your plots from that file! Either create a standalone file (see below for instructions or use the "Export" button in Rstudio), or copy the SVG graphics from the HTML file.

```
# producing an SVG image file
svg("test.svg", width=14, height=7)
plot(1:10)
dev.off()
```

```r
# produce a PDF file
pdf("test.pdf", width=14, height=7)
plot(1:10)
dev.off()
```

Vector graphics with Microsoft Office

If you rely heavily on Microsoft Office for your work, you might find another vector graphic format more useful – the Windows Meta File (WMF). The function is called `win.metafile()` and used in the same way as the `pdf()` function. The problems are that (a) you can only use it on Windows and (b) it does not support transparency.

Remember!

Use SVG format for your plots whenever possible.

5.4 Basic statistics with R

5.4.1 Statistics with R

Exercise 5.6 (P values)**.** P-values are one of the most basic concepts in statistics, and part of the language of science – it is hard to find a scientific paper without any p-values.

- Take a piece of paper or open a text editor and write down a one- or two-sentence explanation of what a p-value is.
- Only when you have done this, read the section "Clarifications about p-values" in this Wikipedia article. Did you get it right? If yes, you are in the minority of scientists.

5.4.2 Descriptive statistics

1

```
x <- rnorm(1000)

# mean
mean(x)
```

```
# median
median(x)
```

```r
# standard deviation
sd(x)
```

```r
# Some basic statistics in one go
summary(x)
```

```r
s <- summary(x)
sum(x < s[2]) / length(x)
```

```r
sum(x > s[5]) / length(x)
```

```r
sum(x > s[2] & x < s[5]) / length(x)
```

```
IQR(x)
```

3.3.1 3.3.2

5.4.3 Simple tests

```
# simulate two vectors with different means
a <- rnorm(15, mean=1, sd=1)
b <- rnorm(15, mean=3, sd=1)

# perform the t-test
t.test(a, b)
```

```
res <- t.test(a, b)
res$p.value
```

```
format.pval(res$p.value, digits=2)
```

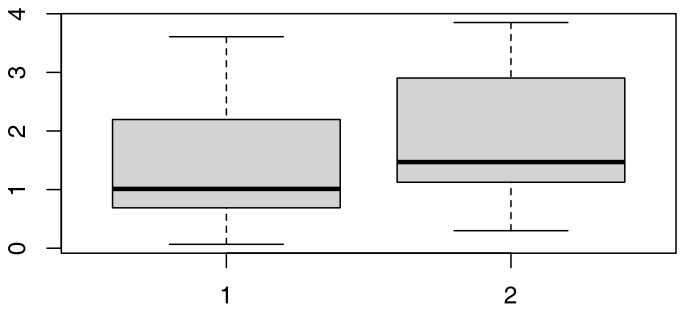
```
"The p-value of the t-test
was `r format.pval(res$p.value, digits=2)`."
```

```
library(broom)
tidy(res)
```

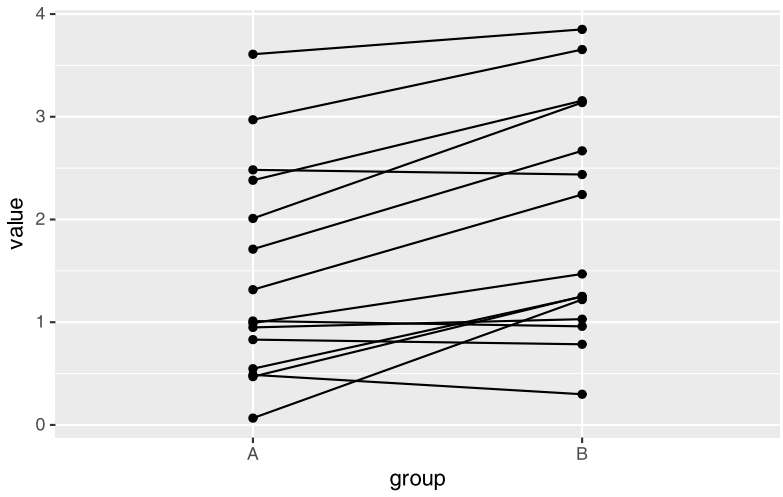
Exercise 5.7 (T-test)**.** Use either the builtin iris dataset or the cleaned-up version of the doctored data set from Day 3. Perform a t-test to compare the sepal length between *Iris setosa* and *Iris versicolor*, as well as between *Iris versicolor* and *Iris virginica*.

```
wilcox.test(a, b)
```

```r
a <- rnorm(15, mean=1, sd=1)
b <- a + 0.5 + rnorm(15, mean=0, sd=.5)
boxplot(a, b)
```



```
df <- data.frame(value=c(a, b),
                 group=rep(c("A", "B"), each=15),
                 id=1:15)
ggplot(df, aes(x=group, y=value, group=id)) +
  geom_point() +
  geom_line()
```




```
t.test(a, b, paired=TRUE)
```

```
library(tidyverse)
tr_res <- read_csv("Datasets/transcriptomics_results.csv")
tr_res <- select(tr_res,
                 Gene=GeneName, Description,
                 logFC=logFC.F.D1, FDR=qval.F.D1)
interferon <- grepl("interferon", tr_res$Description)
significant <- tr_res$FDR < 0.01
table(significant, interferon)
```

```
chisq.test(significant, interferon)
```

Appendix: more stats and visualizations

5.5 PCA and scatter plots

5.5.1 Principal component analysis

3.5.5

-
-
-

5.5.2 The data

```
library(tidyverse)
labdata <- read_csv("Datasets/labresults_wide.csv")
dim(labdata)
```

```
head(labdata[,1:10])
```

```
labdata <- labdata |>
  separate(SUBJ.TP, into=c("SUBJ", "TP"), sep="\\.") |>
  filter(TP == "D1")
```

```
labdata <- labdata |> drop_na()
```

```
# read the meta-data
library(readxl)
meta <- read_excel("Datasets/expression_data_vaccination_example.xlsx",
                   sheet="targets") |>
                   filter(Timepoint == "D1")

combined <- merge(meta, labdata, by="SUBJ")
head(combined[,1:10])
```

```
dim(combined)
```

Check your results!

Check your results frequently. Do you get the number of rows you expect? How does the data look like? Use `dim()`, `head()`, `tail()`, `summary()`, `View()` all the time.

```
combined_mtx <- select(combined, ACA:WBC) |>
  as.matrix()
head(combined_mtx[,1:5])
```

5.5.3 Running the PCA

```
# the actual PCA is just one line!
pca <- prcomp(combined_mtx, scale.=TRUE)
is.list(pca)
```

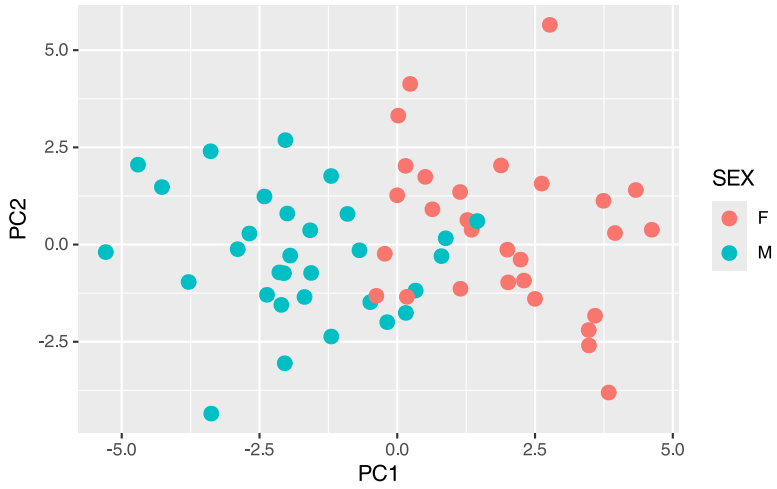
```
names(pca)
```

```
colnames(pca$x)[1:10]
```

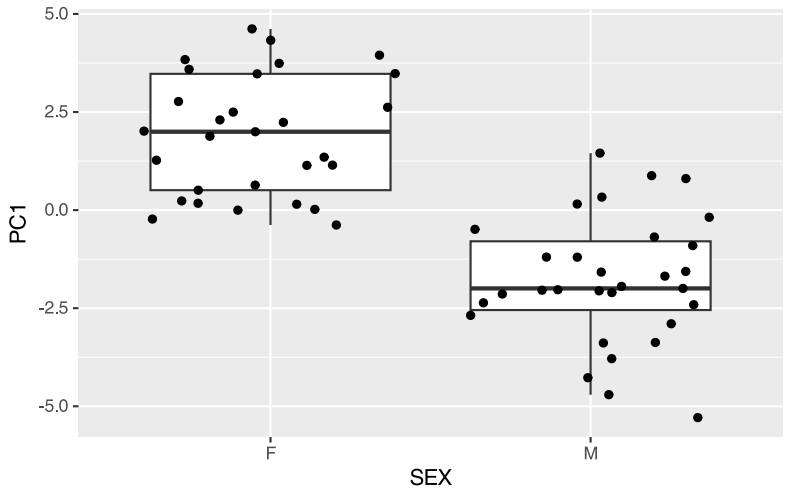
```
dim(pca$x)
```

```
dim(combined)
```

```
library(ggplot2)
pca_df <- cbind(combined, pca$x[,1:10])
ggplot(pca_df, aes(x=PC1, y=PC2, color=SEX)) +
  geom_point(cex=3)
```




```
ggplot(pca_df, aes(x=SEX, y=PC1)) +
  geom_boxplot(outlier.shape=NA) +
  geom_jitter()
```

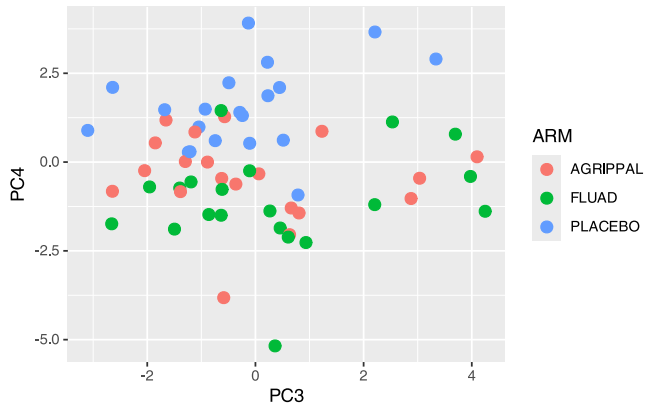


Exercise 5.8 (Vaccines)**.** Repeat the plot above, but instead of `SEX`, use the `ARM` column which corresponds to the study arm, i.e. which vaccine (or placebo) was administered. Can you tell the groups apart? Try it with other components: PC3 vs PC4, PC5 vs PC6 etc. Any luck?

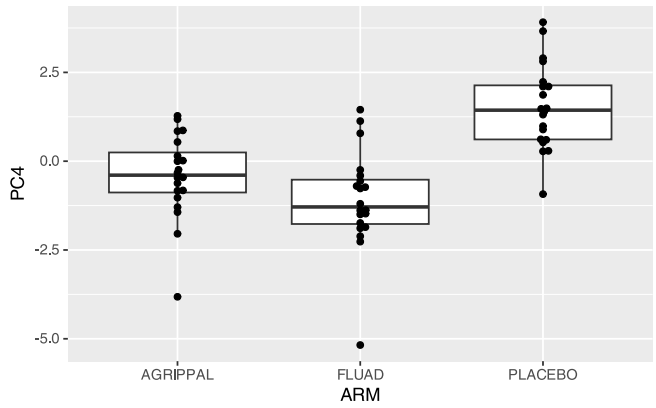
Solution

Yes, the PC4 seems to show differences between the three groups, with Fluad having the lowest values, and Placebo the highest.

```
ggplot(pca_df, aes(x=PC3, y=PC4, color=ARM)) +
  geom_point(cex=3)
```



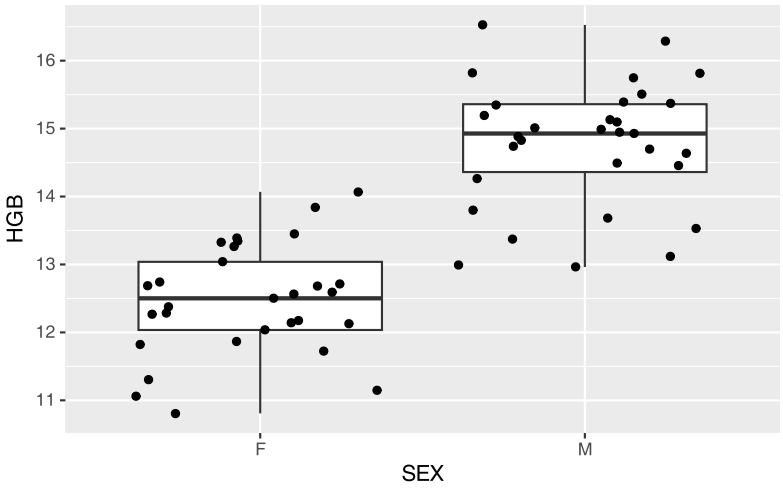

```
ggplot(pca_df, aes(x=ARM, y=PC4)) +
  geom_boxplot(outlier.shape=NA) +
  geom_beeswarm()
```



5.5.4 Interpreting the PCA

```
head(pca$rotation[,1:5])
```

```
pc1 <- pca$rotation[,1]
ord <- order(abs(pc1), decreasing=TRUE)
head(pc1[ord])
```



Exercise 5.9 (Other top loadings)**.** In the table below, you will find three more variables. Check their loadings. Make corresponding box plots and check if the values correspond to what you know about human biology.

Variable	Description
ESR	Erythrocyte sedimentation rate
CREAT	Creatinine
CA	Calcium

Exercise 5.10 (Top loadings for vaccines)**.** Which PC is the most important for separating the vaccine groups? Which variables contribute the most to this component? Can you explain why?

5.6 Afterword

5.6.1 Where to get R packages

CRAN Bioconductor github

- colorDF

- tinytable
- gt

- tidylog

- colorout
- formatdown

- trackdown

- renv

- RColorBrewer

5.6.2 Where to go from here

- StackOverflow

-
  - Hands-On Programming with R

  - R for Data Science (tidyverse)

  - R Cookbook

  - R Graphics Cookbook

  - R markdown: the definitive guide

  - R Workflow

  - R manuals

-
  - The R Book

  - Computational Genomics with R

– Regression and other stories

-

Copilot

5.6.3 Famous last words

> **Get on with R**
>
> Start working with R, right now, for all your projects; for statistics, data management and even preparing reports and documents.
> At first, doing the same task in R will take much more time then the same task in programs you used before. You will feel that you are wasting your time.
> You are not. Sooner then you think, it will pay off.

5.7 Review

-
  -
  -
  -
  -
  -
-
  -
  -
  -
  -
  -
-
  -
  -
  -
  -
-

Appendix: Wide vs long data

Wide and Long format

SUBJ	CA	CRP	WBC
4,081.00	2.31	0.45	12.20
4,368.00	2.14	0.62	6.18
4,601.00	2.26	0.60	5.03
4,224.00	2.25	0.32	7.62
4,323.00	2.04	0.60	5.76

SUBJ	LBORRES_CA	LBORRESU_CA	LABTECH_CA	LBORDATE_CA	LBORRES_CRP
4,081.00	2.31	mmol/L	Frank N. Stein	19072	0.45
4,368.00	2.14	mmol/L	Frank N. Stein	19147	0.62
4,601.00	2.26	mmol/L	Dr. Jekyll	19023	0.60
4,224.00	2.25	mmol/L	Mr. Hyde	18890	0.32
4,323.00	2.04	mmol/L	Dr. Jekyll	19006	0.60

SUBJ	LBTEST	LBTESTCD	LBORRES	LBORRESU	LBORDATE	LABTECH
4,081.00	Calcium	CA	2.31	mmol/L	19072	Frank N. Stein
4,081.00	C Reactive Protein	CRP	0.45	mg/L	18940	Mr. Hyde
4,081.00	White Blood Cells	WBC	12.20	10*9/L	19021	Dr. Jekyll
4,368.00	Calcium	CA	2.14	mmol/L	19147	Frank N. Stein
4,368.00	C Reactive Protein	CRP	0.62	mg/L	18880	Mr. Hyde

•

-
-
-
-
-

Use long format

In general, you should probably always use the long format as the main format for your data. It is easier to work with, and you can always convert it to wide format if you need to.
There are exceptions to this rule – typically when it comes to high-throughput data (like RNA-Seq).

Converting from wide to long:

```
library(tidyverse)
wide <- read_csv("Datasets/wide_example.csv")
wide
```

```
pivot_longer(wide, cols=control:cond2,
  names_to="condition", values_to="measurement")
```

Converting from long to wide

```r
long <- read_csv("Datasets/long_example.csv")
long
```

```r
## not what we wanted!!! Why?
pivot_wider(long, names_from=condition, values_from=measurement)
```

```
## Instead:
pivot_wider(long, id_cols=c(subject, sex),
                  names_from=condition, values_from=measurement)
```

Exercise 5.11 (Converting to long format)**.** Convert the following files to long format:

- `labresults_wide.csv`
- The built-in iris data set (`data(iris)`)
- `cars.xlsx` (tricky! hint: how do you tell which value in the long format belongs to which row in the wide format?)

Clean up and convert to long format (what seems to be the problem? How do we deal with that?):

- `mtcars_wide.csv`

Appendix: More statistics and visualizations

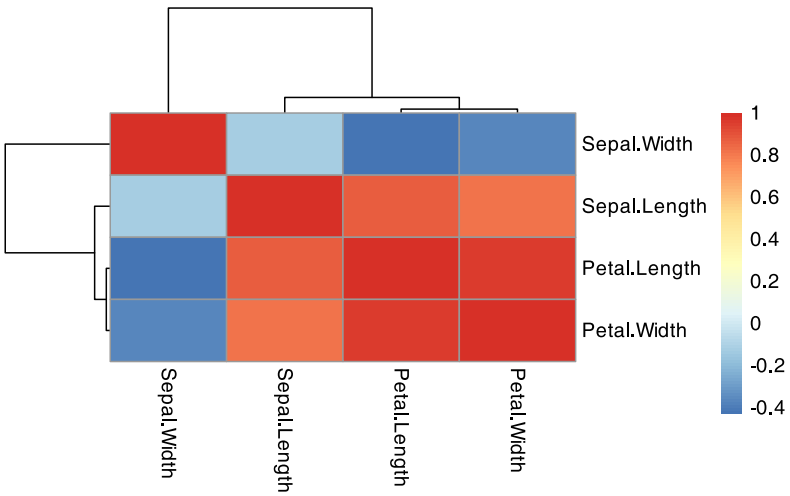
Correlations

```
cor(iris$Sepal.Length, iris$Petal.Length)
```

```
cor.test(iris$Sepal.Length, iris$Petal.Length, method="spearman")
```

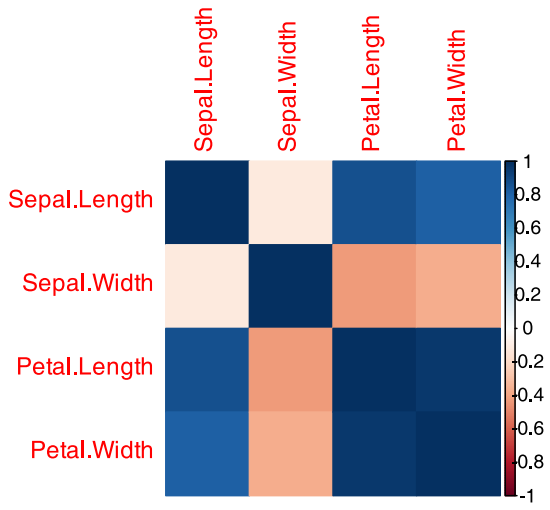
```r
cor(iris[,1:4])
```

```r
library(pheatmap)
M <- cor(iris[,1:4])
pheatmap(M, scale="none")
```

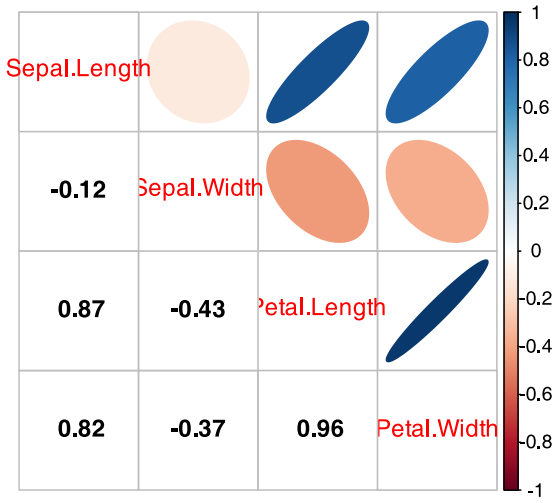


```
library(corrplot)
```

```
corrplot(M, method="color")
```



```
corrplot(M, method="ellipse",
         type="upper", tl.pos="d")
corrplot(M, add = TRUE, type = 'lower',
         method = 'number',
         col="black", diag = FALSE,
         tl.pos = 'n', cl.pos = 'n')
```



Correcting for multiple testing

-

-

```
library(tidyverse)
sv <- iris |> filter(Species != "setosa") |>
  mutate(Species=factor(Species))
pvals <- sapply(1:4, function(i) {
  t.test(sv[,i] ~ sv$Species)$p.value
})
names(pvals) <- colnames(sv)[1:4]
pvals
```

```
p.adjust(pvals, method="bonferroni")
```

```
p.adjust(pvals, method="BH")
```

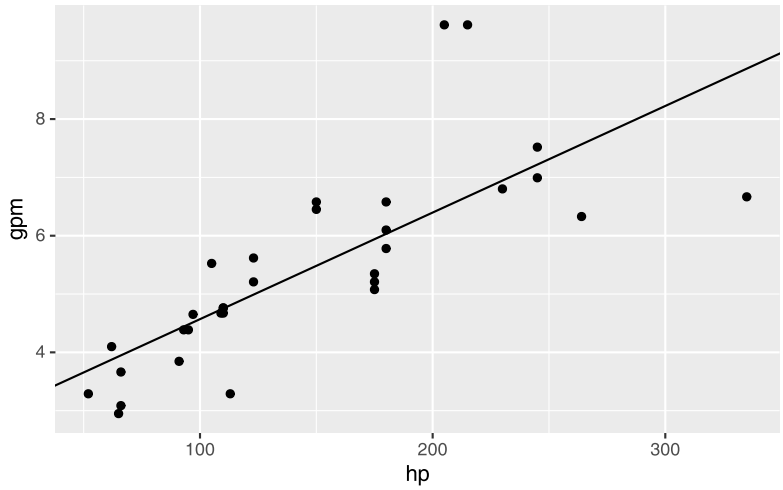
Linear models with `lm()`

Simple linear models

```
mtcars$gpm <- 100/mtcars$mpg
model <- lm(gpm ~ hp, data=mtcars)
```

```r
summary(model)
```

```r
library(ggplot2)
a <- coefficients(model)[1]
b <- coefficients(model)[2]
ggplot(mtcars, aes(x = hp, y = gpm)) +
  geom_point() +
  geom_abline(intercept = a, slope = b)
```




```
model_0 <- lm(gpm ~ 0 + hp, data=mtcars)
summary(model_0)
```

Additional covariates

```
model_2 <- lm(gpm ~ 0 + hp + wt, data=mtcars)
summary(model_2)
```

```r
model_huge <- lm(gpm ~ 0 + hp + wt + qsec + drat + disp + cyl, data=mtcars)
summary(model_huge)
```

```
AIC(model_0)
```

```
AIC(model_2)
```

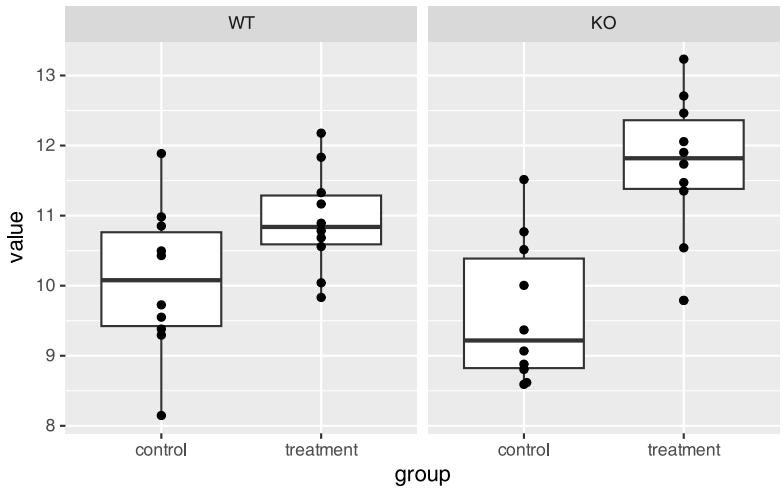
```
AIC(model_huge)
```

```
model_iris <- lm(Sepal.Length ~ Species, data=iris)
summary(model_iris)
```

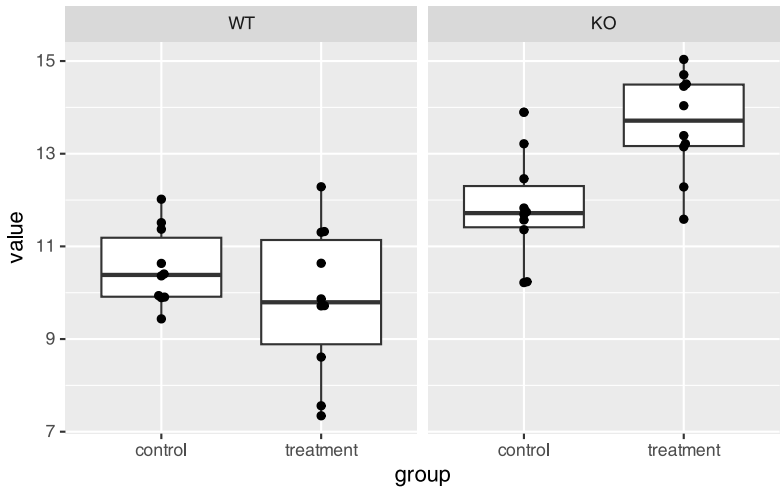
```
anova(model_iris)
```

R Book

Interactions


WT
KO
13
12
11
10
9
8
value
control
treatment
control
treatment
group

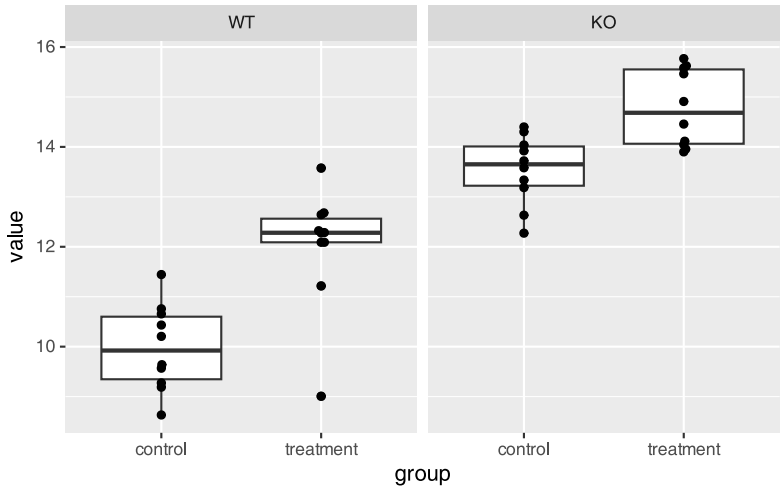
```
dfi <- data.frame(nn = 1:(n*4),
  strain = rep(c("WT", "KO"), each=2*n),
  group = rep(c("control", "treatment"), each = n)) |>
  mutate(strain = factor(strain, levels=c("WT", "KO"))) |>
  mutate(value =
    c(rnorm(n, mean=10, sd=1),
      rnorm(n, mean=10, sd=1),
      rnorm(n, mean=12, sd=1),
      rnorm(n, mean=14, sd=1)))
ggplot(dfi, aes(x = group, y = value)) +
  geom_boxplot() +
  geom_beeswarm() +
  facet_wrap(~ strain)
```



```r
model1 <- lm(value ~ strain * group, data=dfi)
anova(model1)
```

```r
dfni <- data.frame(nn = 1:(n*4),
  strain = rep(c("WT", "KO"), each=2*n),
  group = rep(c("control", "treatment"), each = n)) |>
  mutate(strain = factor(strain, levels=c("WT", "KO"))) |>
  mutate(value =
    c(rnorm(n, mean=10, sd=1),
      rnorm(n, mean=12, sd=1),
      rnorm(n, mean=13, sd=1),
      rnorm(n, mean=15, sd=1)))
ggplot(dfni, aes(x = group, y = value)) +
```

```
  geom_boxplot() +
  geom_beeswarm() +
  facet_wrap(~ strain)
```



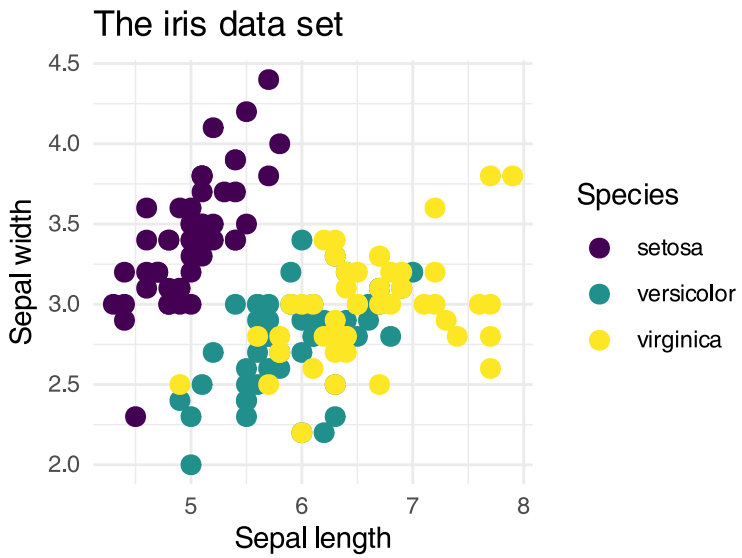

```
model2 <- lm(value ~ strain * group, data=dfni)
anova(model2)
```

Image sizes in R markdown

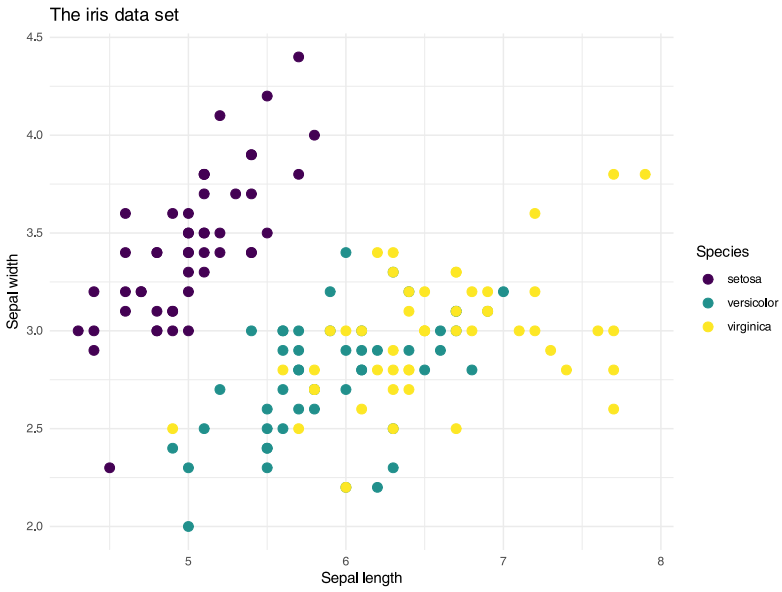
here

5.3.6

```
```{r fig.width=4,fig.height=3}
ggplot(iris,
  aes(x = Sepal.Length,
      y = Sepal.Width,
      color = Species)) +
  geom_point(size=3) +
  scale_color_viridis_d() +
  labs(x = "Sepal length",
       y = "Sepal width",
       title = "The iris data set") +
  theme_minimal()
```
```



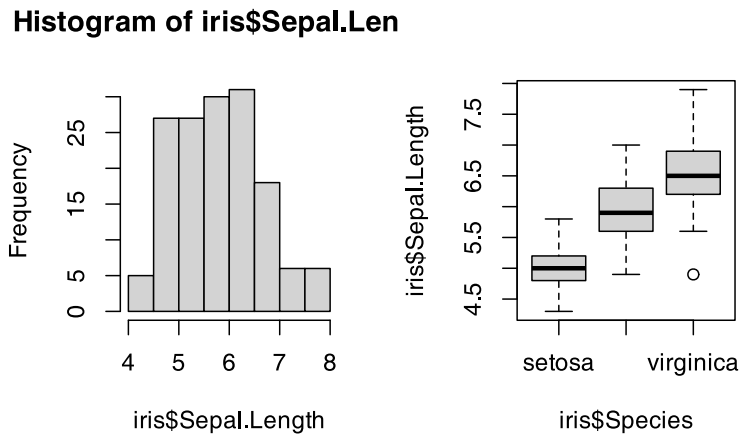

```{r fig.width=8,fig.height=8}
ggplot(iris,
  aes(x = Sepal.Length,
      y = Sepal.Width,
      color = Species)) +
  geom_point(size=3) +
  scale_color_viridis_d() +
  labs(x = "Sepal length",
       y = "Sepal width",
       title = "The iris data set") +
  theme_minimal()
```



Combining several plots into one

Base R

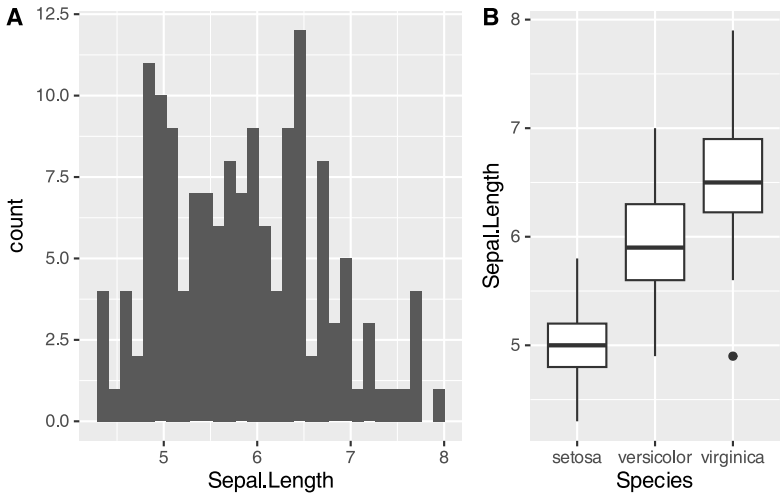
```
par(mfrow=c(1, 2))
hist(iris$Sepal.Length)
boxplot(iris$Sepal.Length ~ iris$Species)
```



ggplot2

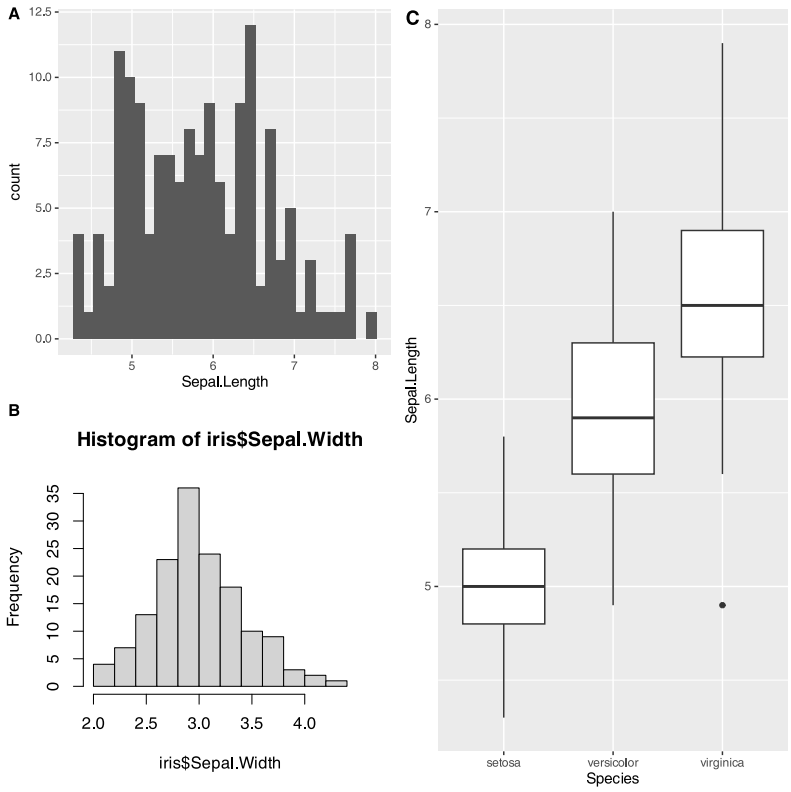
```
library(cowplot)

# we save the plots into a variable
# instead of printing them to the screen
p1 <- ggplot(iris, aes(x=Sepal.Length)) +
  geom_histogram()
```

```
p2 <- ggplot(iris, aes(x = Species, y = Sepal.Length)) +
  geom_boxplot()

# plot both plots with labels in one row
plot_grid(p1, p2,
  ncol = 2,
  labels = c('A', 'B'),
  rel_widths = c(0.6, 0.4),
  label_size = 12)
```



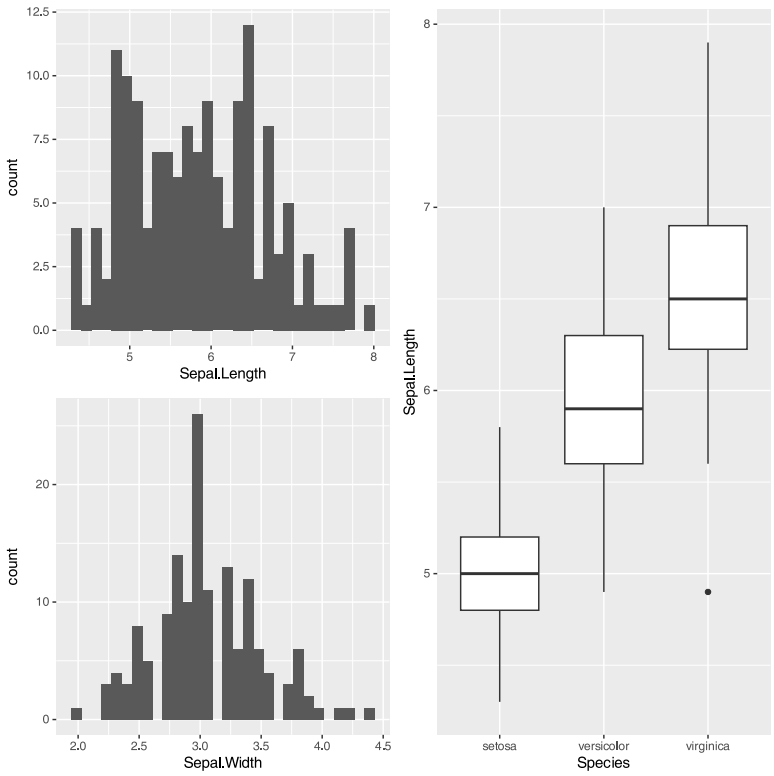

```
# use the formula notation to create a base R subplot
p3 <- ~hist(iris$Sepal.Width)

# create a one column, two-row subplot
p13 <- plot_grid(p1, p3,
  ncol = 1,
  labels = c('A', 'B'),
  label_size = 12)

# put the subplots together
# no need for a label for p13,
```

```
# it already has labels
plot_grid(p13, p2,
  labels = c("", "C"),
  ncol=2)
```



patchwork

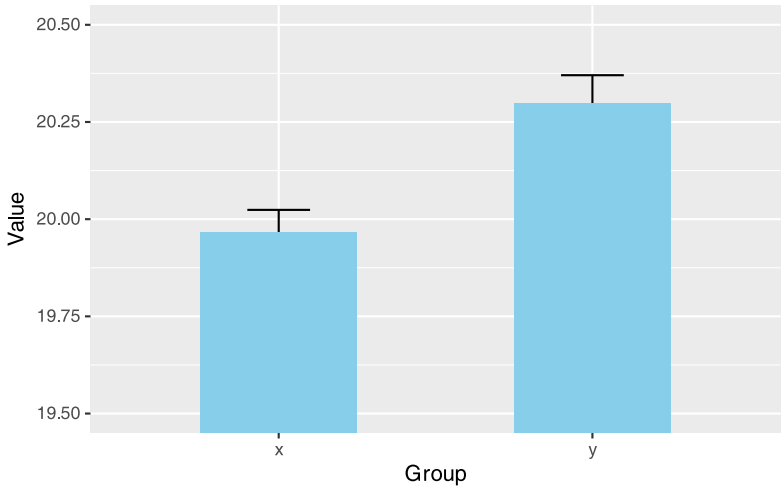
```
library(patchwork)
p3 <- ggplot(iris, aes(x=Sepal.Width)) +
  geom_histogram()
(p1 / p3) | p2
```



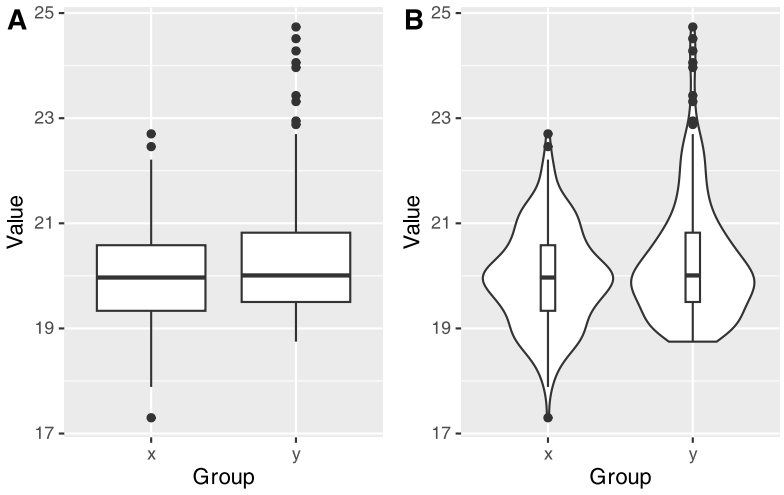
Boxplots and violin plots vs bar plots

```
n <- 250
x <- rnorm(n, mean=20, sd=1)
y <- rbeta(n, 2, 22) * 20  + 18.65
```

```r
df <- data.frame(value=c(mean(x), mean(y)),
                 group=c("x", "y"),
                 sd=c(sd(x)/sqrt(n), sd(y)/sqrt(n)))
ggplot(df, aes(x=group, y=value)) +
  geom_errorbar(aes(ymin = value - sd, ymax = value + sd), width = 0.2) +
  geom_bar(stat="identity", width=0.5, fill = "skyblue") +
  labs(x="Group", y="Value") +
  coord_cartesian(ylim = c(19.5, 20.5))
```



```
library(cowplot)
df <- data.frame(values = c(x, y),
                 group = c(rep("x", n), rep("y", n)))
p1 <- ggplot(df, aes(x=group, y=values)) +
  geom_boxplot() +
  labs(x="Group", y="Value")

# violin plot
p2 <- ggplot(df, aes(x=group, y=values)) +
  geom_violin() +
  geom_boxplot(width=0.1) +
  labs(x="Group", y="Value")

# plot_grid puts the different plots together
# ncol=2 -> two columns
# labels=... -> labels for the plots
plot_grid(p1, p2, ncol=2, labels=c("A", "B"))
```

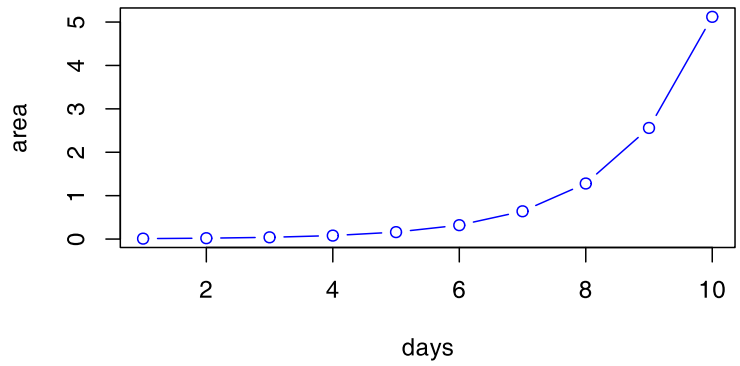

A
B
25
23
21
19
17
Value
x
y
Group

Solutions to Exercises

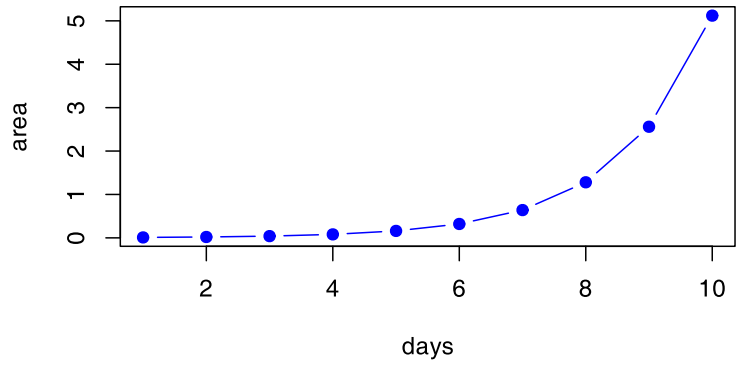
Day 1

Water lilies (Exercise 1.12)

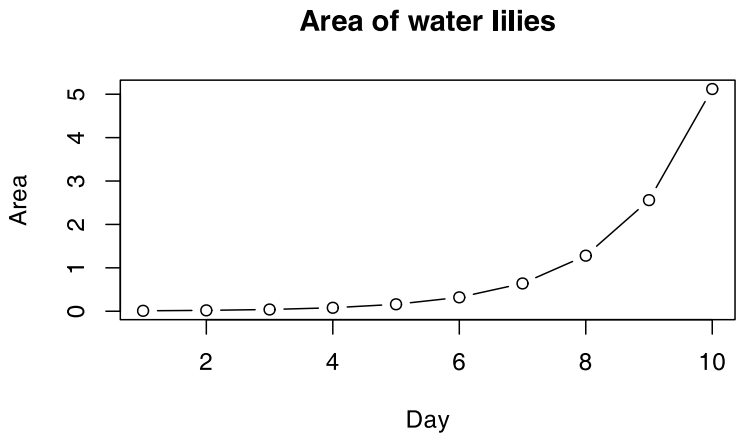
```
# plots with a blue color
plot(days, area, type="b", col="blue")
```



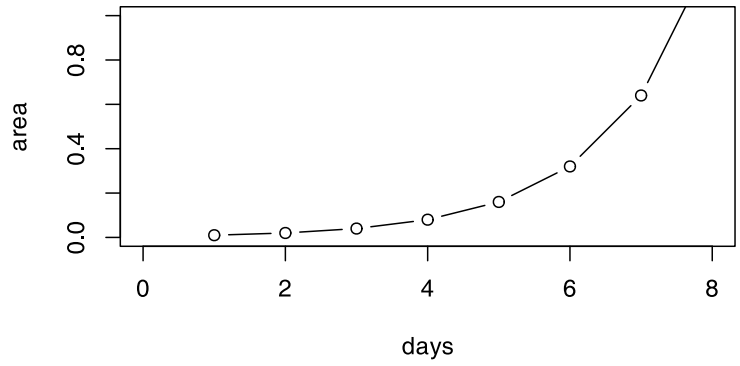

```
# plots with filled circles
plot(days, area, type="b", col="blue", pch=19)
```



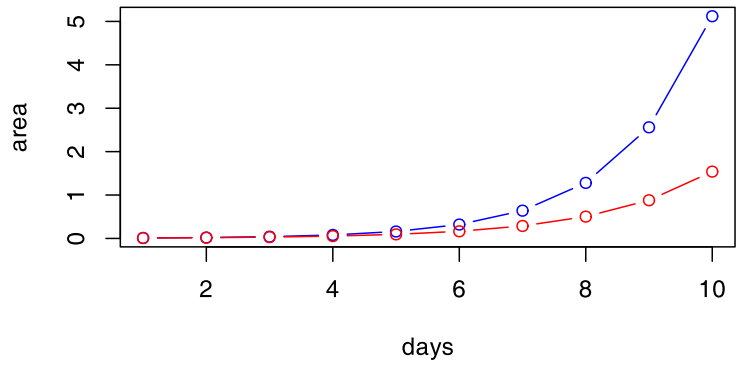

```
# plots with labels
plot(days, area, type="b",
  xlab="Day", ylab="Area", main="Area of water lilies")
```




```
# limits on the y axis and x axis
plot(days, area, type="b",
  xlim=c(0, 8), ylim=c(0, 1))
```




```
area_slow <- 0.01 * 1.75^(days - 1)
plot(days, area, type="b", col="blue")

# add another data series to the plot
lines(days, area_slow, type="b", col="red")
```



Day 2

Matrices (Exercise 2.3)

```
# generate random readouts
readouts <- runif(48)

# create the matrix
res_mtx <- matrix(readouts, nrow=6)

# always check your results!
dim(res_mtx)
```

```
res_mtx[1,] <- NA
res_mtx[,1] <- NA
res_mtx[nrow(res_mtx), ] <- NA
res_mtx[, ncol(res_mtx)] <- NA
```

```
row_names <- c("control", "low", "medium", "high")
row_names <- paste0("Inh1_", row_names)
row_names <- c(NA, row_names, NA)
rownames(res_mtx) <- row_names

col_names <- c("control", "low", "high")
col_names <- rep(col_names, 2)

inh <- rep(c("Inh2_", "Inh3_"), each=3)
```

```
col_names <- paste0(inh, col_names)
col_names <- c(NA, col_names, NA)
colnames(res_mtx) <- col_names
```

```
# wells with inhibitor 3
res_mtx[ 2:5, 5:6 ]
```

```
# inhibitor 1 as well as 2
inh1 <- c("Inh1_low", "Inh1_medium", "Inh1_high")
inh2 <- c("Inh2_low", "Inh2_high")
res_mtx[ inh1, inh2 ]
```

Data frames (Exercise 2.5)

```
mtx <- matrix(rnorm(15), nrow=5)
df <- as.data.frame(mtx)

colnames(df) <- c("X", "Y", "Z")
rownames(df) <- c("a", "b", "c", "d", "e")
```

```
df$A <- rep("A", 5)

df$sequence <- seq(0, 1, length.out=5)

df
```

Day 3

Deaths.xlsx (Exercise 3.5)

```
fn <- readxl_example("deaths.xls")

# one way: use the n_max argument
deaths <- read_excel(fn, skip=4, n_max=11)

# another way: use the excel range specification
deaths <- read_excel(fn, range="A5:F16")
```

Diagnosing `meta_data_botched.xlsx` (Exercise 3.7)

```
botched <- read_excel("Datasets/meta_data_botched.xlsx")
colnames(botched)
```

```
summary(botched)
```

```
library(skimr)
skim(botched)
```

Table 5.4: Data summary

```
age_n <- as.numeric(botched$AGE)
```

```
botched$AGE[ is.na(age_n) ]
```

```
unique(botched$SEX)
```

```
unique(botched$PLACEBO)
```

```
unique(botched$ARM)
```

```
unique(botched[["time point"]])
```

Correcting `meta_data_botched.xlsx` (Exercise 3.13)

```
# either works
botched <- rename(botched, TIMEPOINT=`time point`)
# or
colnames(botched)[2] <- "TIMEPOINT"
colnames(botched)
```

```r
# change to upper case
tp <- toupper(botched$TIMEPOINT)

# replace day with "D"
tp <- str_replace_all(tp, "DAY", "D")

# remove all spaces
tp <- str_replace_all(tp, " *", "")

table(tp, botched$TIMEPOINT)
```

```r
# looks good!
botched$TIMEPOINT <- tp
```

```r
arm <- toupper(botched$ARM)
arm <- str_replace_all(arm, "^P.*", "PLACEBO")
arm <- str_replace_all(arm, "^F.*", "FLUAD")
arm <- str_replace_all(arm, "^A.*", "AGRIPPAL")
arm <- str_replace_all(arm, "^C.*", "PLACEBO")
arm <- str_replace_all(arm, "^G.*", "AGRIPPAL")
unique(arm)
```

```
botched$ARM <- arm
```

```
age <- botched$AGE
age <- str_replace_all(age, "[^0-9]", "")
age <- as.numeric(age)

# check for NAs
any(is.na(age))
```

```
# replace the original column
botched$AGE <- age
```

Day 4

Selecting columns (Exercise 4.1)

4.1

```
# Load the data
library(tidyverse)

results <- read_csv("Datasets/transcriptomics_results.csv")

# what columns are there?
colnames(results)
```

```
# Select the columns that we need
results <- select(results, GeneName, Description, logFC.F.D1, qval.F.D1)
colnames(results) <- c("Gene", "Description", "logFC", "qval")
```

```
results <- select(results, Gene=GeneName,
  Description,
  LFC=logFC.F.D1, FDR=qval.F.D1)
colnames(results)
```

Sorting by last name (Exercise 4.3)

```
persons <- c("Henry Fonda", "Bob Marley", "Robert F. Kennedy",
  "Bob Dylan", "Alan Rickman")

# first, create a vector with last names only.
# basically, remove everything before the last space
# and the space itself
lastnames <- str_replace_all(persons, ".* ", "")
lastnames
```

```
# now get the order for the last names
ord <- order(lastnames)
ord
```

```
# and use it to sort the original vector
persons[ord]
```

Logical vectors (Exercise 4.6)

```
significant <- tr_res$FDR < 0.05
interferons <- str_detect(tr_res$Description, "interferon")
both <- significant & interferons

# how many are there?
sum(both)
```

```
# how many are significant, but not interferons?
sum(significant & !interferons)
```

```
gbps <- str_detect(tr_res$Gene, "^GBP")
sum(gbps & interferons)
```

The merging of two data frames (Exercise 4.9)

4.9

```
# The libraries needed
library(tidyverse)
library(readxl)

# Load the data
labresults <- read_csv("Datasets/labresults_full.csv")
targets <- read_excel("Datasets/expression_data_vaccination_example.xlsx",
```

```
  sheet="targets")

# what columns are there?
colnames(labresults)
```

```
colnames(targets)
```

```
intersect(colnames(labresults), colnames(targets))
```

```
# how many unique subjects are there in each data frame?
length(unique(labresults$SUBJ))
```

```
length(unique(targets$SUBJ))
```

```
# how many are common?
length(intersect(labresults$SUBJ, targets$SUBJ))
```

```
targets <- select(targets, SUBJ, Timepoint, ARM, AGE, SEX)
```

```
labresults <- select(labresults, SUBJ, Timepoint, LBTEST, LBTESTCD, LBORRES)
```

```
joined <- merge(targets, labresults, by=c("SUBJ", "Timepoint"))
dim(joined)
```

```
colnames(joined)
```

Day 5

References

https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1008984

https://doi.org/10.1038/nmeth.2837

https://www.frontiersin.org/journals/genetics/articles/10.3389/fgene.2022.818683/full

https://www.nature.com/articles/s41598-019-56994-8