Psychological Research with R

An R Intro for psychologists whose dog regularly ate their statistics homework.

Anni Tave Overlander

Last update: 2025-02-17

Contents

Welcome!

This book is a work in progress! Many chapters still contain placeholders/ bullet points and will be filled over the next couple of weeks!

Chances are, you are a psychology student and either starting to learn R or looking to refresh your memory. Or perhaps you need to look up a specific step that you *can never quite remember* (or is that just me?). Maybe you also came across this resource by pure chance - lucky you!

In any case I am glad you are here and hope you find both what you were and weren't looking for. This book is based on an in-person introductory R course from the University of Konstanz. I tried my very best to cover all the basics on working with R from the ground up. Following the whole course should enable you to write your very own R Markdown report, taking full advantage of some of the most important and common features of R.

About

...this book

> It's completely based on a course, so you can either look up specific facts or go through each chapter as you would a lesson.

This book is quite *opinionated*, meaning I included all the lovely things that *I* like to work with. You might prefer other packages and that is completely okay - I still appreciate you reading my suggestions. As with anything in life, a lot can be learned from other peoples approaches to things.

Each chapter will end with a "Wrap-up and Further Resources "-section, i.e. a little collection of facts you should now know and some links for further reading. Also, in most chapters you will find

some exercises or hidden code snippets, so you can test your skills as you go along.

Important: I will present you with a lot of different packages that I believe will make your life easier in this book. Please make sure that you have them installed if you want to try out the code. You can typically install any package by typing `install.packages("packagename")` - substituting packagename for the name of the package, e.g. dplyr - into your R console and hitting Enter. Some packages may not be installable this way, in which case I will explicitly mention how to install them where they are first used!

...the author

I am Tave and I am currently working on my Ph.D. in Research Methods, Assessment and iscience. This online book is a little side project that is quite near and dear to my heart.

In my experience, statistics and more so statistics programs can easily induce panic-like states in psychology students. And also in my experience, that can change over the course of one semester tops. Programming in R is a lot less scary than many may think and most of all, it can and *should be fun*! It is a weird, powerful language and can assist you with many everyday tasks.

I hope to alleviate some of the aveRsion over the course of this course and help you see R's advantages.

Please feel free to contact me if you have any questions or comments! You can reach out via e-mail: annika-tave.overlander@uni.kn or you can submit a GitHub issue over at the repository for this book: https://github.com/the-tave/psych_research_in_r.

Part I

First Steps

Chapter 1

Intro and Installation

If you have no prior experience with R, the different names can get a little bit confusing. There is R, but there is also R Studio and the two serve different purposes in your workflow. So in the following I will try to make a clear distinction and guide you through the installation of both R and R Studio.

1.1 What is R? What is R Studio?

First of all, R is a programming language for statistical and data analysis.

It is open source, which means anyone can contribute - an many many many people do. Like a lot of other programming languages, R can be used with the functions that it understands without any further instructions - we call this base R - or you can use *packages* - think of them as new tricks that you can teach your R.

That might all sound intimidating at first but more than anything else it means that means that R follows instructions. The slightly tricky part is learning how to give those instructions.

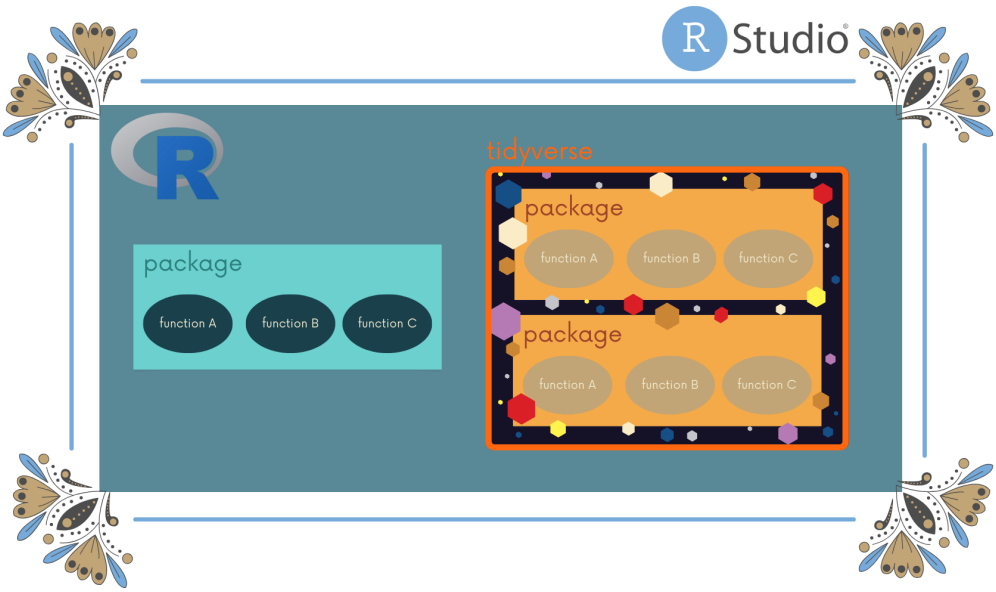


Figure 1.1: R and R Studio as a schematic. R Studio is the 'pretty frame' for R, which can and should be used with packages that in turn contain different functions. The 'tidyverse' is included as a special teaser - it is a meta-package which has many packages that follow a similar workflow and logic.

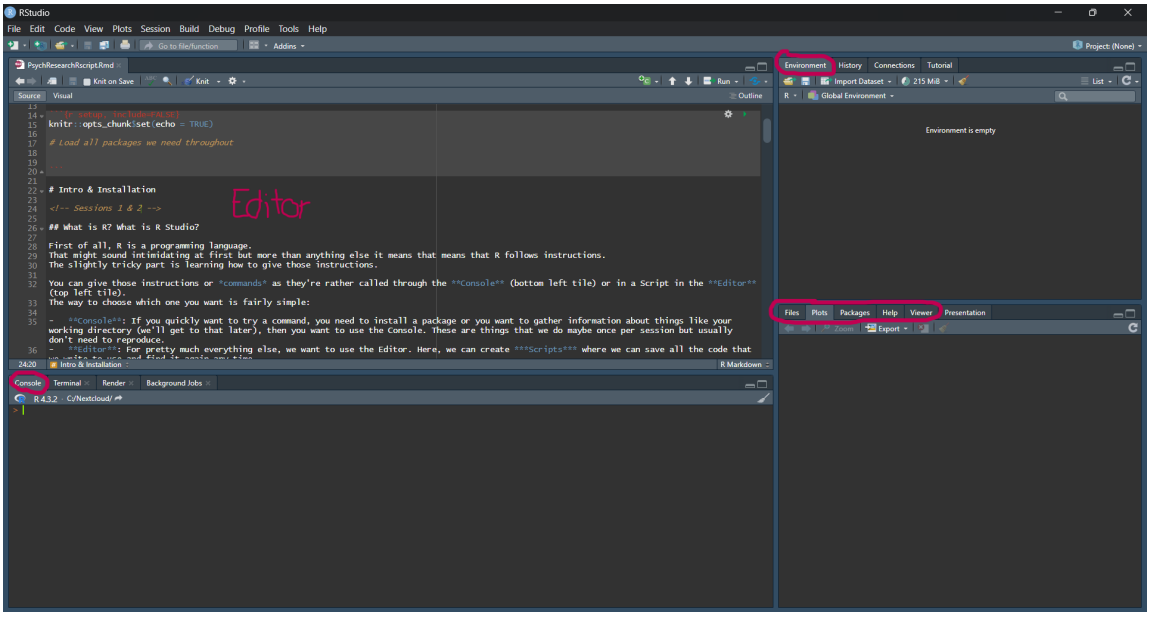


Figure 1.2: R Studio - typical layout

1.2 Installing R and R Studio

- Download & Install the newest R version 4.4.2 (2024-10-31 ucrt) at https://cloud.r-project.org/
- Download and install R Studio at https://posit.co/download/rstudio-desktop/

R as a language can be used on its own. However, it is not very modern, quite hard to use and frankly just no fun. That's why we use R Studio as a user interface to run R. Think of it as dipping your fingers in a pot of ink to write versus using a pen and paper - you will still write with the same ink, but the whole process is just nicer.

1.2.1 Workflow

You can give instructions or *commands* as they're rather called through the **Console** (bottom left tile) or in a Script in the **Editor** (top left tile). The way to choose which one you want is fairly simple:

- **Console**: If you quickly want to try a command, you need to install a package or you want

to gather information about things like your working directory (we'll get to that later), then you want to use the Console. These are things that we do maybe once per session but usually don't need to reproduce.

- **Editor**: For pretty much everything else, we want to use the Editor. Here, we can create ***Scripts*** where we can save all the code that we write to use and find again any time.

For most things - especially during the learning phase - it makes sense to write a Script in the Editor in order to be able to save and access the work. To do so, simply click the  button and choose 'R Script'.

Once we have a new script created, we can learn some basic things that R is capable of and save it to check out later. Let's try to use R as a calculator first. In the *code chunk* below you will find some mathematical operations and their respective output. Here as well as in other places that show R code (such as forums, blogs or help pages) you can recognize a code chunk by its gray background and mono space formatting. Output is presented with the `[1]` at the start of the line.

Basic Math

```
1+2
  [1] 3
2-3
  [1] -1
3*4
  [1] 12
4/3
  [1] 1.333333
5^2
  [1] 25
```

Assigning Values

```
a <- 3
b <- 4
```

```
a-b
# [1] -1
a*b+5
# [1] 17
```

In these code chunks, R is essentially being used as a calculator to perform basic math. While we want to make use of all the more powerful functions of R, it is important to grasp the basics and be able to use arithmetic for our purposes. Next to the basic mathematical operators, there are many nifty mathematical functions, such as `sqrt()` - *square root*, `sum()` or `pi`.

We also added **variables** containing values (here a and b hold values 3 and 4 respectively) that can be used in the calculations just like the values they contain. As they only contain numbers, the variables a and b are called **numeric**. We can check this property of a variable, e.g. `a` using the function `class(a)`, which gives us "numeric" as output.

Usually when we fire up R, we don't just want to work with single values numeric but with **data frames** or **vectors**  that can contain different classes of variables and several values respectively.

Vectors

```
v <- c(6, 7, 8)
class(v)
# [1] "numeric"
d <- c("sunny", "rainy", "foggy")
class(d)
# [1] "character"
```

Data Frame

```
e <- data.frame(v, d)
class(e)
# [1] "data.frame"
e
```

```
#   v     d
# 1 6 sunny
# 2 7 rainy
# 3 8 foggy
```

Notice that we did not just add all values one after the other, but followed a certain notation that begins with the function `c()`. The c stands for "combine" or "concatenate" and tells R that all following values belong to the same variable. There are several ways of adding variables to a data frame, but the `data.frame()` command is the simplest.

> All values in a variable should have the same class. Go ahead and try out `hm <- c(3, "sunny", 5.2)` and check the class. What happened - did you expect that?

When we have our data in a neat data frame, what we usually want to do is access either certain **rows** or **columns**. To do so, base R uses square brackets `[]` behind the name of a data frame to indicate "take this data, but only certain rows/columns/cells". Remember: The brackets understand the first input as rows and the second as columns - *rows right away.*

Square Bracket Indexing

```
e[1, ]
#   v     d
# 1 6 sunny
e[ , 1]
# [1] 6 7 8

weather <- e[ , 2]
weather
# [1] "sunny" "rainy" "foggy"

e[1, 1]*e[2, 1]
# [1] 42
```

Our data set `e` contains three numbers in column c and different strings in column d. We can select just the first row, or just the first column or either of the other rows and columns that we have in the data. We can also re-assign the values, e.g. to a new variable named "weather" or perform calculations on single cells in the data frame - only if they contain numerics, of course.

> With square brackets we can only index rows and columns that are present in the data. That maybe sounds obvious, but can easily lead to confusion because of error messages! What happens when you try to index `e[,3]`?

While it is common practice to work with data frames and edit them according to our tasks and needs, using square brackets and base R can get a bit humdrum. Luckily, many many people have developed many many packages that contain different functions, helping us in most tasks that we will need to tackle!

1.3 Installing Packages

Figure 1.3: Hexagon Package logos

Most packages are available on CRAN - the Comprehensive R Archive Network. That being the case you can easily install the package you want to have with the command `install.packages("packagename")`. It is important to note that the package name must be in quotes for the installation, while loading it into a script works without quotes using `library(packagename)`.

Packages usually serve quite specific purposes, e.g. ones we will later get to know are `dplyr`, with which data handling is made a lot easier and more intuitive and `ggplot2`, which allows us to create beautiful, publication-ready plots and visualizations. What is special about these two, among some others, is that they were developed by the same person (Hadley Wickham), and are made available in a sort of "meta package" - the `tidyverse`. Installing and loading the `tidyverse` makes the functions from many different packages available at once. This is quite convenient when we want

to use a lot of those packages in the same session or script, but does also take a longer time to load and is sometimes not actually necessary.

To use this package of packages, please install the `tidyverse`, using `install.packages("tidyverse")` in the console and then load it into your script (or, again, directly in the console, bottom-left) with `library(tidyverse)`. You can test whether it works by running `iris %>% pull(Sepal.Length) %>% mean()` in your console. We will get to know the syntax in depth another time, but just so you know what's going on: This line of code takes the data set `iris`, which is included in R by default, "pulls" the variable `Sepal.Length` out of the data and runs the `mean()` function on it, to calculate the average sepal length, which should be 5.8433333.

Posit Cloud

If you would like to be able to use R with several devices or simply want to start out and see what all the fuss is about, you can use the free online version of R Studio over at https://posit.cloud/. There, you can do most anything that you can also do locally on your own machine, online. It's a great way of working with R online without any commitment.

However, if you plan to really learn R or also if your teacher plans you really learn R, I would recommend following the above steps to use it your own computer. Some calculations or operations can require a bit more computation power that might be slower via the internet.

Wrap-Up & Further Resources

R is a powerful language, R Studio is the user interface we use with it

R can be a calculator and perform basic and advanced math

We mostly work with variables and data frames

Packages make working with R easier and more fun!

Tadaa Data: R für Psychos (german)

Kaggle: Intro to R

Tidyverse

Chapter 2

R Basics & Error Messages

2.1 Basics

R is an object-based language. The great advantage of that is we can assign values, vectors, text, matrices or almost anything else to variables and access them more easily later. To do that, we use a left arrow ← like so: `x <- c(1, 2, 3)` Go ahead and try it out! If you had too much fun assigning variables, you can remove them again from your work environment using the `rm()` command in the Console. So, if you assign the values 1, 2 and 3 to `x`, you can type `rm(x)` into the console, hit enter and x has disappeared from your environment!

When you are working on a script, you can write and save all of your code. You can also execute any line of code with `ctrl + enter` in the editor as you go along to test your code. When you just

want to run something once or test your code, you can enter it in the console where you just need to press `enter`/ `return`.

In your script, comments can and should be added to your code using the `#`. It is considered good practice to use comments generously, which will also help you to understand whats going on when you look at something a couple of days/weeks/years later and keep and overview!

Basic Functions

Assign values & look at a variable

```
a <- c(1, 2, 3, 4)
b <- c(5, 7, 9, 11)
a
# [1] 1 2 3 4
```

Text aka *strings* need to be put in quotes so it can be recognized as such

```
some_text <- c("amazing", "wow")
```

Mean/average

```
mean(a)
# [1] 2.5
```

Standard deviation

```
sd(b)
# [1] 2.581989
```

Minimum

```
min(b)
# [1] 5
```

Maximum

```
max(b)
# [1] 11
```

Add values

```
a+b
# [1]  6  9 12 15
```

Find the sum of all values in a vector

```
sum(b)
# [1] 32
```

Find the length of a vector

```
length(a)
# [1] 4
```

Define a sequence

```
v <- 3:9
v
# [1] 3 4 5 6 7 8 9
```

Find the range of a vector

```
range(v)
# [1] 3 9
```

Define several sequences in one vector

```
d <- c(10:15, 20:25)
d
#  [1] 10 11 12 13 14 15 20 21 22 23 24 25
```

Now that we know how to assign values and how to find some basic stats on single vectors, we will look at how to extract specific values from vectors. We call this indexing and in R we can use square brackets to do so.

Find the first value in a vector

```
d[1]
# [1] 10
```

Find the value where a vector has a specific value - this is especially interesting when we want to compare two vectors

```
d[d==10]
# [1] 10
a[d==10]
# [1] 1
```

Less convoluted: Find the index where a vector has a specific value

```
which(d==12)
# [1] 3
```

You can practice these functions on your own and definitely check out the Exercises section!

> **Brainteaser** : We define two variables in the following way:
>
> ```
> e <- 1:5
> f <- 2:5
> ```
>
> Let's say we want to access the number 5 in both variables. How come `e[5]` works but `f[5]` does not?

Now, you may be wondering why I skipped the letter c and whether you should be taking the word of someone who clearly does not know their ABCs. The reason is quite simple: You already saw the function `c()` in the last chapter, which is used to create variables that are larger than one entry[1]. It stands for combine and it was used in the code above to create the variable `d`, containing two sequences. So, while you can use single letters as variable names for placeholders, it is generally not a great idea to use `c` - if we can avoid confusing for ourselves, we want to do it. There is a whole section on how to name your variables in Section **??** so that future-you won't be mad.

[1] Technically, that is also not true as you just saw that we can create continuous sequences with a colon, i.e. `1:3` is 1, 2, 3.

2.2 Logic

Next to numeric variables like `a` and `b` that hold numbers, and string variables that hold `"text"`, there are also so-called *boolean* variables. They can only hold the values **TRUE** or **FALSE** and are either assigned specifically by you, the user, or result from basically asking R a yes-or-no question. For example:

Is 3 smaller than 7?

```
3 < 7
# [1] TRUE
```

Is 3 smaller than 2?

```
3 < 2
# [1] FALSE
```

Now, these examples are fairly trivial but boolean variables using logic can be really helpful, e.g. for filtering data. You can find the mean of some variable for a specific group by telling R your criteria and the operation you want to conduct on the filtered data. Let's say we asked some students about their gender and their skills with R. If we want to know the average R skills of male students we would tell R: "If the gender of a participant is male, include them in the mean value for R skill". So our code would look something like this:

```
mean_r_skill_m <- mean(data$r_skill[data$gender == "male"])
```

Of course, if you try to run this code you will probably encounter one of the errors described in the next section, because we do not currently have a dataset called `data`. But you can try out the principle with one of the Exercises! Depending on the analysis you need, you can use any of these so-called "logic operators":

```
x <- 4; y <- 5
x < y # smaller than
# [1] TRUE
x <= 4 # smaller/ equal
# [1] TRUE
```

```
x > y # greater than
# [1] FALSE
x >= 3 # greater/ equal
# [1] TRUE
```

```
x == y # equal to
# [1] FALSE
x != y # not equal to
# [1] TRUE
z <- TRUE
z
# [1] TRUE
!z
# [1] FALSE
```

You can find a concise overview over all of these operators in the Toolbox.

2.3 Reading Error Messages

Even the most advanced R coder will encounter the occasional error message. While they are quite helpful and usually lead to being able to solve problems, it can be challenging to learn how to properly read the message in order to actually understand what the problem is. Therefore, we will look at some of the most common wordings to decipher what R needs so it can understand what we want to do.

Think of error messages not as discouraging faults of your program but rather as invites to help you help R understand what you are trying to achieve.

Error: unexpected 'X' in "Y"

This message is fairly straightforward: R expected a certain kind of symbol at the place where we put 'X' and is now confused because our input did not match the expectation.

For most Europeans, for example, the most commonly used decimal separator will be a comma.

Since R is an american-built language, however, it expects decimals to be separated with a decimal point while commas indicate separate inputs to a function. Therefore, when we accidentally input a decimal as '3,141', we will receive the error message *Error: unexpected ',' in "3,"* because R expected the input to be something like '3.141'.

Fix: Replace the 'x' in "Y" with something that R can understand. Or alternatively: Make sure that we meant the input like that.

Error: object 'A' not found

This, too, is pretty understandable message: The object that you are trying to access and use cannot be found by R. The most common cause for this error message is probably simply that you misspelled the object name somewhere. R is case-sensitive, so maybe object *A* was defined as object *a*? Maybe the object *vector* is spelled *vcteor* in your function?

Moreover, sometimes our thoughts are two steps ahead of our code. When figuring out how to get a program to work, sometimes we presume that a variable exists just because we need it and forget to define the variable up front. The same goes if we try to access a variable in a data frame that was maybe defined in a different data frame or as its own vector - it simply cannot be found because we are having R look in the wrong place.

Fix: Make sure the object name is spelled correctly and was defined prior to using it in a function, calculation or elsewhere.

This error message also often appears with "function xyz not found". In this case, we probably forgot to load the package first, which contains that function. Thus, the fix will likely be to figure out which package contains the function we are trying to use and load it with the `library()` command.

Error: R does nothing after running a command

This is not an error message but it is still very common, especially at the beginning of your learning journey. Usually, there is a `>` symbol at the beginning of your console input line, which indicates that R is ready to run some code. However, if R is "unfinished" with a command, you see a `+` instead. This happens when R can't work with the command because there is something missing,

which in most cases, will be a closing parenthesis.

Fix: Click in your console and use the `Esc` button to cancel the command. Then, look at the code you were trying to run and see if there is some closing statement such as `)` missing and try again.

The pipe operator `|>` or `%>%`

Next to missing closing parentheses, a classic source for this behavior is the so-called pipe operator, created with either `|>` (native R pipe, available since R version 4.1.0) or `%>%` (tidyverse pipe from the `magrittr` package). It will be used a lot starting in Chapter **??** and it is probably one of my personal favorite R features. The pipe allows you to use previous operations in the next one easily and clearly. In reality this relates to not drowning in parenthesis and everything in the `tidyverse` is designed to be used with a pipe.

Let me show you what I mean: We will create a variable that contains some numbers and we want to find the average value rounded to two decimal spaces. We already know the `mean()` function; to round a number we use the `round()` function and define how many decimal places to round to after we input the value we want to round.

```
# Code without pipe
round(mean(c(5, 9, 2)), 2)
# [1] 5.33


# Code with pipe
c(5, 9, 2) |> mean() |> round(2)
# [1] 5.33
```

As you can see, both lines of code will give the same result but the pipeline follows a much more intuitive way of coding and allows you to write your code as you would think of the analysis. As opposed to that, in the first line you need to first think of rounding, then of the mean and then last input your actual values, which seems very backwards. Also, as I mentioned, you can see that there are already some parentheses next to each other, which would be really easy to forget and not receive any output.

Generally, I wholeheartedly recommend getting familiar with piping very early on when learning R. However, don't forget that R expects input after each pipe. If you accidentally end a line of code with a pipe, you will encounter the behavior where R just does nothing and you basically have a staring battle of who will give in first. The same fix will work here: Hit the Esc button, look at the code you were trying to run and figure out if there might have been a pipe that just went nowhere.

> When you need help figuring out how a function works, there are several ways to get it.
> Try typing ¿'|>" into your console and hit enter!

```
knitr::include_url("./img/nerfect.webp")
```

So, always keep in mind: Mistakes happen to everyone. Error messages can be really frustrating but they really are meant to be helpful. Sometimes R has a hard time understanding what's wrong, so if an error message seems vague, think of it as your chance to play detective and figure out what happened. Also, I promise you that mastering the art of reading error messages can be tedious, but it is very worth it. Once you get a feeling of how R tries to identify errors, it gets a lot less disheartening. It can actually be quite wholesome to see a big, intimidating error message that makes absolutely no sense at all, just to find a teeny, tiny typo in your code which works once you fix it.

Exercises

Logic

We want to find out whether the average Petal Length and Sepal Length of iris flowers are different from each other.

Use what you know to "ask R" if those values are unequal!

Hint: You will need logic, the mean()-function and square brackets. Use **names(iris)** to find all variable names in that data set.

Solution

- Square bracket indexing with the variable name in quotes
  `iris[ , "Sepal.Length"]`

`iris[ , "Petal.Length"]`

- now we add the mean function around both like this: `mean(iris[ , "Sepal.Length"])`
  `mean(iris[ , "Sepal.Length"])`
- Finally we compare the two with the != operator: `mean(iris[ , "Sepal.Length"]) != mean(iris[ , "Petal.Length"])`
  TRUE

Create a vector

Create a vector x with the numbers from 50 to 100 and 150 to 200.

Find its mean, standard deviation and its range.

What is the 77th number of vector x?

Solution

- `x <- c(50:100, 150:200)`

- `mean(x), sd(x), range(x) %>% round(2)`:
  125, 52.38, 50, 200
  - `%>% round(2)` takes the numbers and rounds them up to two decimal points
- `x[77]`: 175

Code with problems

Below is some code that has a few problems.

Try to identify them and how they might be fixed. Feel free to test them out if you are not sure!

```
Library(greatpackage)
```

Solution

- `library` should not be capitalized
- "greatpackage" does not exist and can thus not be loaded

```
mean(coolvariable)
```

Solution

coolvariable was not defined previously

```
a <- c(1, 3, 6, 7
```

Solution

the command is not finished, it needs a closing parentheses

```
b <- c(2, 4; 6, 8)
```

Solution

we need all commas to separate numbers in a vector, not a semicolon

Wrap-Up & Further Resources

Functions work with input inside round brackets, e.g. c(1, 2, 3)

a point **.** is a decimal separator in numbers; a comma **,** seperates input in functions

Logical operators compare data, e.g. $7 > 6$ would output TRUE

`#` allows comments in the code

Errors should be invitations to make your code more understandable for R

→ the better we understand the problem, the better we can fix it!

StackOverflow

Discovering Statistics Using R Book by Andy Field, available from KIM

Chapter 3

Best Practice

" ~~All~~ Many roads lead to Rome"

You can achieve most things in many different ways - *best practice* refers to the best/ easiest/ clearest way of working with R. Some of the things you will read about might not make sense to you intuitively and that's ok. Realistically, there are some mistakes that everyone must make for themselves. When you do, I just want you to remember that here is some advice that you have either heard before or will read up on now to get out of the pickle you may be in! Maybe you would like to bookmark the Toolbox to always find an overview of nifty tips, tricks and hacks?

3.1 Naming Conventions

R is a so-called *object-oriented* language. What that means for us is mostly that all our data exist as "objects" as far is R is concerned. Just like in real life, we can *do* stuff with those objects now, which is called using a *function* or *command*. Which brings me to the importance of proper names for all your variables, data sets, functions… Everything.

If we go to the market and I tell you to get me an apple, you will probably be confused if there are many types of apples, maybe different colors, maybe different breeds. Basically, if I don't specify which apple I want, you are not able to pick out the right one. The same is true for R: If you try to call a function but fail to specify what object to use the function on, R is confused and throws

you an error.

Now, this may seem pretty obvious but it is a pretty common source of errors, especially during the "steep phase" of the learning curve. In R, `abc` is a different object than `ABC`, which is different from `a_b_c`, which is different from `A.B.C`. All of these variants are possible ways of naming your objects in R. However, it makes everyone's life significantly easier to stick to some naming-guidelines.

- Preferably use **lower-case** variable names, e.g. `gender` instead of `Gender` or `GENDER`.
- Preferably use an **underscore** to differentiate between different words in your object names if necessary, e.g. `music_preference`.
- **Avoid using numbers** in your names because likely either you or R will get confused with this at some point, e.g. `raw_data` instead of `data1`[1].
- Use **abbreviations** where useful, e.g. `rt` instead of `reaction_times`.
- Use **names that will still make sense** to you in the future, i.e. avoid names like `asdf_data` or `blibblobfundata`. The best case scenario would be that your variable names also makes sense to other people if they try to understand your code!

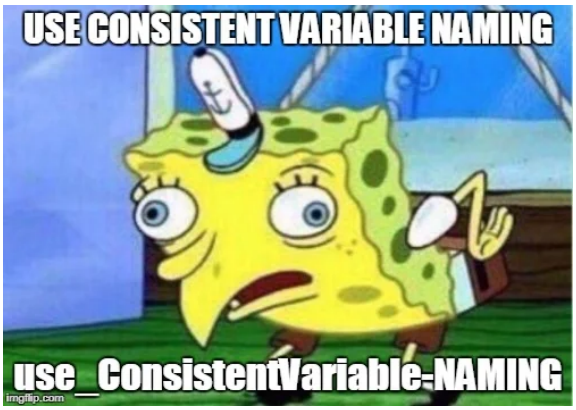
I will admit that some of these pieces of advice are more opinionated personal experience than objective facts. The more you work with variables and maybe also code from other people, you may form your own opinions on what the best naming conventions are for you. I strongly suggest finding a way that works for you and sticking with it. As I described above, I personally will try to stick to *lower_snake_case* to name my variables because it is usually very clear, easy to read by humans and computers and would also be usable in any other programming language.

Another thing that can take some time to get used to is avoiding white spaces in file names and any other names, for that matter. R has problems finding files with names such as "My file with a really specific name.bib", which can easily be avoided by sticking to snake_case: "My_file_with_a_really_specific_name.bib". The same goes for variables: In my first class, we discovered that it is possible in R to set a variable name with a space. Just because it's possible does not mean anybody should do it. Ever.

You can try it out: Enter `"hi there" <- 5` in your console to assign the value 5 to a variable

[1] Additionally, the 1 (one) and the l (lower-case L) can look very similar, which makes thing just even worse when trying to figure out an error.

named `hi there`. You will see it appear in your working environment, but accessing the variable is virtually impossible. If you just type `hi there` in the console and hit Enter, R will give you an error à la *"unexpected symbol"*. But if you type `"hi there"`, it will echo the text back to you.



https://www.reddit.com/r/ProgrammerHumor/comments/6stwag/can_you_stick_to_any_naming_convention/

Here is a list of positive and negative naming examples:

- `great variable`
  - Spaces can technically work but are a hassle to access
- `Areallysuperinsanelyverylongvariablename`
  - Variables are supposed to *reduce* workload
- `Howaboutthis$`
  - Special characters should be avoided
- `age_grouped`
  -
- `GenderRecoded`
  -
- `aMAZING~vARIABLE~nAME`
  - Please just no.

3.2 Exercise: Creating Data

Imagine we are going to measure some test scores at a school that are supposed to reflect the kids' IQ (mean 100, sd 15). In order to prepare for data analysis, we want to simulate what the data might look like beforehand.

We are going to measure their age and IQ score and we will be assessing class 7a and 7b.

Create **two data frames** - one for each class. They should each contain **n = 20 entries** (for 20 kids) and **2 variables i.e. age and IQ score**. We assume **age is a random number from 12 to 15** and **IQ follows normal distribution with mean = 100, sd = 15**.

Afterwards, add a variable to code the **class** to each data frame and add them together **underneath each other**.

Which functions should I use?

The functions you will need are *data.frame, sample, rnorm, $, rbind* and *str*. You can read help on any function by typing a question mark before the function name in your console and hitting Enter, e.g. `?rnorm`.

Solution

```
# Create the data frames
scores7a <- data.frame(age = sample(x = 12:15, size = 20, replace = T),
                       iq = rnorm(n = 20, mean = 100, sd = 15))

scores7b <- data.frame(age = sample(x = 12:15, size = 20, replace = T),
                       iq = rnorm(n = 20, mean = 100, sd = 15))
# Notice anything about the code?

str(scores7a); str(scores7b)
# 'data.frame': 20 obs. of  2 variables:
#  $ age: int  14 12 12 14 12 13 13 13 15 14 ...
#  $ iq : num  108.5 129.6 84.9 120.5 85.3 ...
# 'data.frame': 20 obs. of  2 variables:
```

```
#  $ age: int  14 13 13 13 12 15 13 13 14 13 ...
#  $ iq : num  127.1 68.3 111.9 118.5 110.6 ...

# Add information about the class
scores7a$class <- "7a"
scores7b$class <- "7b"

# create big dataframe
allscores <- rbind(scores7a, scores7b)

# Check the overall data frame for plausibility
str(allscores)
# 'data.frame': 40 obs. of  3 variables:
#  $ age  : int  14 12 12 14 12 13 13 13 15 14 ...
#  $ iq   : num  108.5 129.6 84.9 120.5 85.3 ...
#  $ class: chr  "7a" "7a" "7a" "7a" ...
table(allscores$class)
#
# 7a 7b
# 20 20
```

3.3 Working Directories

A working directory corresponds to the folder on your computer that you are **working in**. It is similar to when you open up a program such as Word and then locate a file that you wish to open there - you need to find the folder on your machine where that file is located. With R, we usually either just open up R Studio or we will directly open a script. Commonly, the working directory will either be the default folder (e.g. "Documents" on Windows) or if you open up a script from a specific location, it might also be set to that folder automatically. Vice versa, if you work on a script in R Studio and save it, it will saved in the folder that is currently set as the working directory.

You can check your current working directory by entering `getwd()` in the console. It will output a so-called file path that "explains" which folder your are working in. In the default view of R Studio you can also find the "Files" tab in the lower right corner and it will show the contents of your current working directory. If you want to change the working directory you can use the command `setwd()` and enter a file path in the parentheses.

> Try it out: Execute the getwd() command in the console. Which folder is R working in - is it the one you expected?

While this way of structuring your work in folders and directories is very common, it can pose some issues down the line. Usually, we will work with our own data files (see Chapter **??**) and read them into our script to use them for analyses. If they happen to be in a different folder than our working directory, we will need to use the complete file path to tell R where to find this data file. When we now ask someone else to try out the code - either because we need help or because we are handing in an assignment - they will raise an eyebrow because chances are that they do not have the specified file path available on their machine.

> [1] "C:/Users/Tave/great_projects/my_specific_folder_name"

The same goes for us if we ever want to use a different machine, e.g. for work. Moreover, it is quite easy to forget which folder you are currently working in, which means it can get hard to later locate your scripts again.

This may all sound like quite the hassle, but luckily there is a simple solution built right into R Studio that will make self-organization easy.

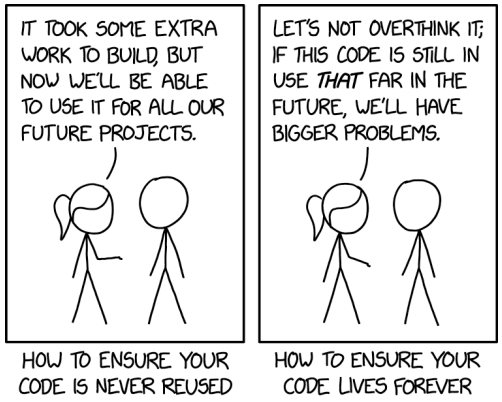
3.4 R Projects 

A project in R is essentially a bookmark to a specific folder on your computer. It lets you organize your projects very clearly, so you can e.g. create a folder for your thesis where you save the data you collected and the R scripts to wrangle that data. When you open the R project it automatically sets your working directory to the folder you put the project in. It also makes it easier for you to use so-called *relative file paths*, allowing you not to specify a full file path when you want to reference a data file or maybe a BibTex file that contains citations for a paper or thesis. With a

relative file path you can use a period as a placeholder for "current folder/working directory" and then just specify the path from there.

Let's try that out under `File` → `New project...`

This is also great for sharing your code and projects: If someone else has the same data file in their R project then they will be able to use your scripts containing just the relative path and be able to execute the code.



https://xkcd.com/2730/

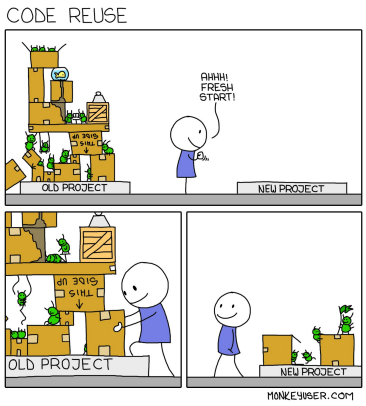
Next to all advantages with sharing as well as creating reproducible code, working with R projects will also help you in keeping your workflow organized. In the following, you can try to create your own R project and see how to use the relative file path!

Exercise

1. Create a subfolder called "data" in your R project.
2. Download the file "mindfulness_data.Rds" from GitHub and save it in that folder.
3. Now try to load it into your script and check whether it worked by executing this code:

```
mindful <- readRDS("./data/mindfulness_data.Rds") # relative filepath
str(mindful)
# tibble [601 x 6] (S3: tbl_df/tbl/data.frame)
#  $ observing : num [1:601] 3.9 3.8 5 3.5 3.7 3.2 3.9 3 3.6 3.1 ...
#  $ describing: num [1:601] 3 2.9 3.5 3 3.3 3.1 2.9 2.9 3 3 ...
```

```
#  $ accepting : num [1:601] 3.7 4.6 1.7 3.9 4 3.3 1.7 3.1 2.2 4.1 ...
#  $ acting    : num [1:601] 2.7 3.8 2.8 3 3.3 4 2.4 3.4 3 3 ...
#  $ age       : num [1:601] 38 26 53 56 18 34 70 24 31 43 ...
#  $ gender    : Factor w/ 3 levels "male","female",..: 2 1 2 1 1 1 2 2 1 1 ...
```



https://www.reddit.com/r/ProgrammerHumor/comments/11a1fqi/code_reuse/#lightbox

3.5 Make R your own

So far, you have learned about many important aspects of getting used to a productive and efficient workflow. However, in my experience the best basis for a good workflow means nothing if it is not really fun to use said flow. Therefore, I want to introduce you to a really nifty feature in R Studio: customizing the appearance.

`Tools` → `Global Options` → `Appearance`  Appearance

Because honestly? it's fun, and if a program is fun you will want to use it more. And practice is is the best way of learning any new skill.

There are many different "themes" for R Studio to choose from. You can control not only what the interface looks like but also how code is displayed. This can be especially helpful if you have a hard time differentiating colors - you can look for a theme that makes it easier for you! Most themes also have some rules for coloring numbers differently than text differently than function calls an so on. So it is not just a nice add on but can actually be really helpful and improve accessibility.

Personally, I work really well with the `Merbivore` theme - what is your favorite?

Wrap-Up & Further Resources

Stick to naming conventions for R objects and related files

Make sure your working directory is correct

Use R projects to organize your scripts easily

Make R fun and appealing to use FOR YOU!

R Projects: A quick overview

Starting your R projects

RStudio Projects and Directories (rather thorough){target="_blank"}

Some ideas on improving your workflow (This can seem a bit advanced, but try reading through anyway and see what sticks with you. Building good habits at the beginning of your learning journey will make your life a lot easier down the line!)

Working with Version Control

…can sound scary and daunting to learn. I believe this to be a more advanced feature which is a bit much for a (non-programmer) beginner, so I will mention it last. However, using version control systems such as GitHub can be really great to keep track of your work, share with others and also ask for help.

If you want to start learning about Git and GitHub, I suggest starting your journey directly on the GitHub Docs webpage. For those who may already have some experience, here is a nice blog article bringing up some of the slips and falls one might take when working with GitHub: Hack your way to a good Git history (Maëlle Salmon).

(PART) Into the `tidyverse`

Chapter 4

Data munging with `dplyr`

In this chapter you will learn about working with packages, which I find similar to teaching your R new tricks. I will also introduce you to the `tidyverse` - a collection of packages and why I strongly suggest to familiarize oneself with a bunch of them. Spoiler: It will make your life a lot easier.

4.1 Working with packages

First off, when you stick to the metaphor a package being a collection of tricks to teach your R, then you will teach the tricks once and remind R that it knows the tricks every time you want to use them. In more real life terms, you will install a package once and load it in each new session. This is important to remember because installing packages is one of the few things you will *only* want to do in your console - NOT in any of your scripts. Nothing bad will happen if you do, but it takes a very long time and it will test both your own patience and that of your computer.

Generally, packages can be installed with the `install.packages()` function. It needs the package name in quotes as input in the parentheses, e.g. `install.packages("dplyr")`. This means the package is now on your computer but you still need to make it available. The easiest way to achieve that is to load the whole package using the `library()` command. This will usually be done inside a script where you want to use functions from that package. Later in this chapter, when we look at dplyr functions on the `iris` data in Section **??**, you will see a code chunk as a prototype of a

typical beginning of an R script.

While `library()` makes all functions from a package available, you can also only load specific functions right when you want to use them. This works with the notation `package::function()` and is particularly useful if (1) the package is extremely large, takes a long time to load and/or you really only need the one function or (b) the function name you want to use is not unique, i.e. there are several packages that contain a function of that name. For example, the function *filter* exists in more than one package, so you might use `dplyr::filter()` in your code to make explicit that you want to use the `filter()` function from the `dplyr` package.

When this is the case, you will get the message "The following objects are masked from…" in the console when you load the package. This is also good to keep in mind if you run into unexpected errors - maybe you meant to use the function from another package but R is using the one you loaded last?

In the interest of creating reproducible code, it can also make sense to use the `package::function()` notation, so you can explicitly show others which function you used.

At a glance:

- Install packages with `install.packages()` only once in the console!
- Load packages with `library()` every time you load a script or start a new session
  - Usually in the script, but also in the console
- Specific function from a package: `package::function()`
  - Use if you just need that one function once and don't want load the entire package
  - Or to explicitly show the according package
  - Or the function name is also in other packages (e.g. "filter" is usually `dplyr::filter`, but sometimes `stats::filter`)

4.2 The tidyverse

> The tidyverse is an opinionated **collection of R packages designed for data science**. All packages share an underlying design philosophy, grammar, and data structures. (tidyverse.org)

So packages are pretty cool: They usually come from people who had a specific problem, found a solution and decided to share it with others so they can more easily fix new problems. Of course everyone has a different preference on how to work, which is why not every package and function will be intuitive to everyone. That is also why the definition of an "opinionated collection of R packages" is so fitting because the `tidyverse` is a framework consisting of many different packages that the authors find intuitive and useful.

Figure 4.1: Hex Logos

Incidentally, I agree and I will show you why. If you figure out at some point that your prefer another way - that's great, but the tidyverse is a solid starting point for most data science needs in psychology.

The structure of the `tidyverse` packages is very clear and they each serve a distinct purpose. Taking a real-life workflow as inspiration, we will first get to know `dplyr`, a package for cleaning up data and editing everything to your needs in this chapter and later get to know `ggplot2` as a package for data visualization in Chapter **??**. One of the big advantages of the `tidyverse` packages is that the can be combined very easily, creating easy-to-read and -write workflows that you will probably still be able to understand one year after you wrote it.[1]

You can install the whole `tidyverse` at once, which will take a little while as it contains a lot of individual packages. Generally, I would recommend installing them once, but loading packages

[1] Being able to read your own code after a while is already challenging but try making sense of code that someone else has written!

individually when you need them, because also loading the bulk of packages takes a lot of time.

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

4.2.1 What is dplyr?

The `dplyr` package is probably the best and most important package in R *imho*. It is a powerful tool for editing data in data frames and a great way to keep your workflow clear and reproducible. It has very intuitively named functions that on their own already serve most purposes that you will commonly need to properly work with your data. If you haven't already, now would be a good time to install `dplyr` and load it into a practice script:

```
install.packages("dplyr") → library(dplyr)
```

In the following, we will use a data example to get to know some of the most important functions. Try to follow along and remember that you can always get more info on a function with a question mark in front of the function name in the console (here best to add the package name), e.g. `?dplyr::filter`. If you want more general information on the whole package you can use the command `browseVignettes(package = "dplyr")` in the console. Also check the further resources and an overview of the most important dplyr functions.

4.3 Intro to dplyr

Figure 4.2: Types of Iris Flowers

We introduced the `iris` data set in Section **??** - a native R data set that you can just access without needing to load anything. It contains information on iris flowers of different species and we will use the `glimpse()` function from `dplyr` to get a first overview of the data we have at hand:

```
dplyr::glimpse(iris)
# Rows: 150
# Columns: 5
# $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4.~
# $ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.~
# $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.~
# $ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.~
# $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, setosa, s~
```

Imagine...

When working with "real" data, aka survey or experimental data, we will usually need to reshape, filter and generally edit the data a bit. This may sound a bit sketchy at first, but we do not want to fake or change any data, we just want to ensure that we have the highest-quality data set to work with that we can.

In the example we will be working with, we have the following three goals for the data:

1. We only want data from species "virginica".
2. We are interested in sepal length.
3. We want to adjust variable names.
4. We later also want the species to be capitalized in our data & to add a new binary variable for whether a flower's petals are longer than 5.5 cm (1) or not (0).

Using `dplyr` functions, we can easily achieve all these goals in a few lines of code. We will go more into depth

```
library(dplyr)


iris_virginica <- iris %>% # create new data set as copy
  filter(Species == "virginica") %>% # 1. only virginica
```

```
  select(Petal.Length, Species)  %>% # 2. select columns
  rename(plength = Petal.Length) # 3. rename variable/column


str(iris_virginica)
# 'data.frame': 50 obs. of  2 variables:
#  $ plength: num  6 5.1 5.9 5.6 5.8 6.6 4.5 6.3 5.8 6.1 ...
#  $ Species: Factor w/ 3 levels "setosa","versicolor",..: 3 3 3 3 3 3 3 3 3 3 ...
```

4.3.1 Filter

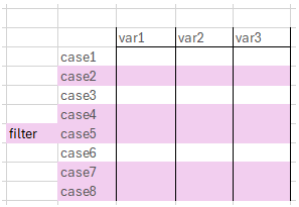


Figure 4.3: filter scheme

- Filters data by a **value in a variable**
  - With filter we **keep only certain rows** in our data
- Needs **logical operators** as input
  - "keep only cases where variable x has value abc"
  - Can also be used with the ! to drop certain cases
  - Drop NAs: `data %>% filter(!is.na(variable))`

4.3.2 Select

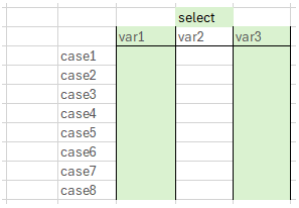


Figure 4.4: select scheme

- Selects **variables based on their names**
  - With select we **keep only certain columns** in our data
- Needs **variable names** as input
  - "keep only variables with this name"
  - Can also be used with the ! to drop certain variables
- Cave: if we want to extract one variable to calculate something, you will want to use `pull()` instead
  - Check `iris %>% select(Species) %>% class()` and `iris %>% pull(Species) %>% class()`

4.3.3 Rename

This is probably the most intuitively named function: It renames a variable in a data frame. As input the function takes your new variable name, an equal sign and the old variable name. This order is important, so if you run into an error, make sure that you are using *new_variable = old_variable*. Also, you do not need quotes around your variable names.

4.3.4 Mutate

- We also want the species to be capitalized in our data
- We also want to add a new binary variable for whether a flower's petals are longer than 5.5 cm (1) or not (0)
  - for this, we use the `ifelse()` function:
  - Needs a logical as first input, then what to do if it's true, last what to do if it's false
- Mutate takes a **variable name as input** (existing or new) and some **function or calculation** to be done on that variable

```
iris_virginica <- iris_virginica %>%
  mutate(Species = toupper(Species), # 4. change content of Species and add new binary variable bigboi
         bigboi = ifelse(plength > 5.5, 1, 0))


table(iris_virginica$Species)
#
```

```
# VIRGINICA
#        50
```

4.3.5 group_by & summarize

- With the dplyr-workflow, we can easily output group statistics
- Keywords: **group_by()** & **summarize()**
- The code is built like any other dplyr workflow with pipes (%>%) in between each:
  1. Define/ name the data frame
  2. group_by(*variable_name*)
  3. in summarize, define a name for the statistic, use a =, use a function like mean() for the measure

```
str(Orange) # use different default data set
# Classes 'nfnGroupedData', 'nfGroupedData', 'groupedData' and 'data.frame':    35 obs. of  3 va
#  $ Tree         : Ord.factor w/ 5 levels "3"<"1"<"5"<"2"<..: 2 2 2 2 2 2 2 4 4 4 ...
#  $ age          : num  118 484 664 1004 1231 ...
#  $ circumference: num  30 58 87 115 120 142 145 33 69 111 ...
#  - attr(*, "formula")=Class 'formula'  language circumference ~ age | Tree
#   .. ..- attr(*, ".Environment")=<environment: R_EmptyEnv>
#  - attr(*, "labels")=List of 2
#   ..$ x: chr "Time since December 31, 1968"
#   ..$ y: chr "Trunk circumference"
#  - attr(*, "units")=List of 2
#   ..$ x: chr "(days)"
#   ..$ y: chr "(mm)"

Orange %>%
  group_by(Tree) %>%
  summarize(m_age = mean(age),
            m_circumference = mean(circumference),
            n = n())
```

```
# # A tibble: 5 x 4
#   Tree  m_age m_circumference     n
#   <ord> <dbl>           <dbl> <int>
# 1 3      922.            94       7
# 2 1      922.            99.6     7
# 3 5      922.           111.      7
# 4 2      922.           135.      7
# 5 4      922.           139.      7
```

Exercises

Data Munging

Imagine we want to edit the iris data set for our colleagues from the US, who are interested in the petal width of the **setosa** and **versicolor** species.

1. Create a new dataset from `iris` with a meaningful name
2. `select()` the variables of interest
3. `filter()` the species that we want
   - Use %in% to filter by more than one value, or think about a reverse approach…!
4. `rename()` the Petal.Width variable to be named pwidth
5. Use `mutate()` to add a new variable named "pwidth_inch", which contains the petal width in inches
   - Calculation: pwidth / 2.54 (2.54 cm = 1 inch)

Solution

```
iris_twospec <- iris %>%
  select(Petal.Width, Species) %>%
  filter(Species %in% c("setosa", "versicolor")) %>%
  # or: filter(Species != "virginica")
  # or: filter(Species == c("setosa" | "versicolor"))
  rename(pwidth = Petal.Width) %>%
```

```
  mutate(pwidth_inch = pwidth / 2.54)


head(iris_twospec, 8)
#   pwidth Species pwidth_inch
# 1    0.2  setosa  0.07874016
# 2    0.2  setosa  0.07874016
# 3    0.2  setosa  0.07874016
# 4    0.2  setosa  0.07874016
# 5    0.2  setosa  0.07874016
# 6    0.4  setosa  0.15748031
# 7    0.3  setosa  0.11811024
# 8    0.2  setosa  0.07874016
```

Figure 4.5: Hadley Wickham, developer of the tidyverse

Group Statistics

Follow the structure to group the iris data set by Species and output a summary with the mean values of all four other variables in the data. Also include the grouped n.

Solution

```
iris %>%
  group_by(Species) %>%
  summarize(m_plength = mean(Petal.Length),
            m_pwidth = mean(Petal.Width),
            m_slength = mean(Sepal.Length),
            m_swidth = mean(Sepal.Width),
            n = n())
# # A tibble: 3 x 6
#   Species    m_plength m_pwidth m_slength m_swidth     n
#   <fct>          <dbl>    <dbl>     <dbl>    <dbl> <int>
# 1 setosa          1.46    0.246      5.01     3.43    50
# 2 versicolor      4.26    1.33       5.94     2.77    50
# 3 virginica       5.55    2.03       6.59     2.97    50
```

Piping Hot %>%

You already heard about the pipe operator in Section **??**. The pipe - available as a native R pipe |> or a `tidyverse` pipe from the `magrittr` package %>%[2] - takes the input from the left-hand to the right-hand. In simpler terms, it takes the operation you did before the pipe and uses is as input for the function after the pipe. This keeps our code readable and tidy and allows us to keep edits in separate lines, but we still only have to run one command for every task we need It's easy to add new commands by using another pipe operator, or to leave edits out but commenting out the respective line of code.

It is important to say that most things can be achieved with or without pipe and you may prefer different versions for different tasks. You have seen a basic example of the use of the pipe operator in Section **??**, so now we will look at a more sophisticated example: We will perform similar operations as we did above and code them first with pipe and then without.

[2]The two pipes can pretty much the same operations - for specific differences see https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/

```r
# With pipe
iris_edit <- iris %>%
 filter(Petal.Length > 3) %>% # only big petals
 select(Species, Petal.Length, Sepal.Length) %>%  # only those columns
 mutate(random_calculation = Petal.Length * Sepal.Length) # calculate random new variable

# Without pipe - two versions
iris_edit2 <- filter(iris, Petal.Length > 3)
iris_edit2 <- select(iris_edit2, Species, Petal.Length, Sepal.Length)
iris_edit2 <- mutate(iris_edit2, random_calculation = Petal.Length * Sepal.Length)

iris_edit3 <- mutate(
  select(
    filter(
      iris, Petal.Length > 3
    ),
    Species, Petal.Length, Sepal.Length
  ),
  random_calculation = Petal.Length * Sepal.Length
)

# Are the three versions equal?
all.equal(iris_edit, iris_edit2) & all.equal(iris_edit, iris_edit3)
# [1] TRUE
```

As you can see, all three version produce the same output and all are technically correct. However, in the second example we keep re-assigning new edits to the same data frame (would be even more annoying but possible to assign each edit to a new data frame) and the third version is just quite the nightmare to read if you ask me. Also think about this:

> In the part with pipe we used the `iris` data to edit and assigned it to `iris_edit`. In version two without pipe we used `iris` in the first line, but afterwards we used

`iris_edit2` for assignments and edits. Why is that?

If we were to use `iris` in each line, it would overwrite the last edit every time.

We can also profit from the pipe in base-R or with a larger combination of different functions. Usually, the pipe helps us not to drown in parentheses when conducting analyses. Let's e.g. calculate a mean value after filtering by Species setosa.

So without a pipe, it might look like this:

```
round(mean(iris$Petal.Length[iris$Species == "setosa"]), digits = 2)
# [1] 1.46
```

Using all pipes, the code would look a lot longer but still easier to read.

```
iris %>%
filter(Species == "setosa") %>%
pull(Petal.Length) %>%
mean() %>% round(digits = 2)
# [1] 1.46
```

As a third alternative, we might also use a combination of the approaches, such as:

```
round(mean(filter(iris, Species == "setosa") %>% pull(Petal.Length)), digits = 2)
# [1] 1.46
```

As with most things in R and programming, different approaches may be easier in different situations, so it has merit to think outside the box at times!

> Brainteaser : We want to edit the iris data to rename a Petal.Length to pl and mutate it to be multiplied by ten. Why does the first version work, but not the second one?

```
# Version 1
iris %>%
  mutate(Petal.Length = Petal.Length * 10) %>%
  rename(pl = Petal.Length) %>%
  glimpse()
```

```r
# Version 2
iris %>%
  rename(pl = Petal.Length) %>%
  mutate(Petal.Length = Petal.Length * 10) %>%
  glimpse()
```

In the second version, we rename the variable Petal.Length to pl and afterwards try to access it with its old name.

Wrap-Up & Further Resources

Packages make working with R easier

`dplyr` is a powerful tool for editing data (*select, filter, mutate…*)

The pipe %>% makes your code clearer and "follows the thought process"

R for Data Science Book

dplyr vignette

Pipe operator

Cheatsheet: https://github.com/rstudio/cheatsheets/blob/main/data-transformation.pdf

YouTube: 20 R Packages that you should know - you for sure don't need them all, but there are some nice inspirations for working with packages in there

https://programmerhumor.io/debugging-memes/well-played-3/

Figure 4.6: Which idiot...

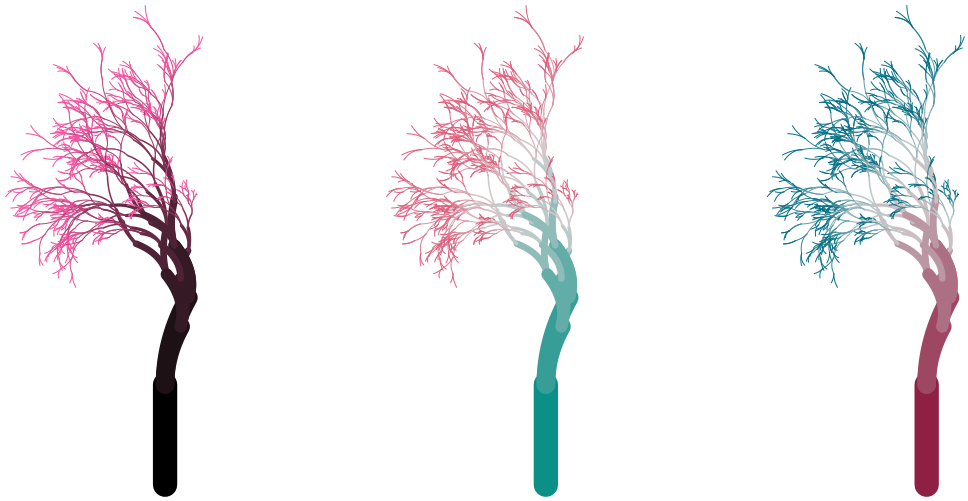
Chapter 5

Pretty plots with `ggplot2`

As you may have gathered by now, I strongly believe that learning (R) should be fun in order to motivate people to keep going. I also strongly believe that being able to literally look at your data is immensely helpful in understanding the type of statistics you can calculate with it, what the distribution looks like and what generally we are dealing with. Therefore, we will learn how to make our data visible to both us and others and get to know `ggplot2` as powerful data visualization package from the `tidyverse`

5.1 Data Visalization

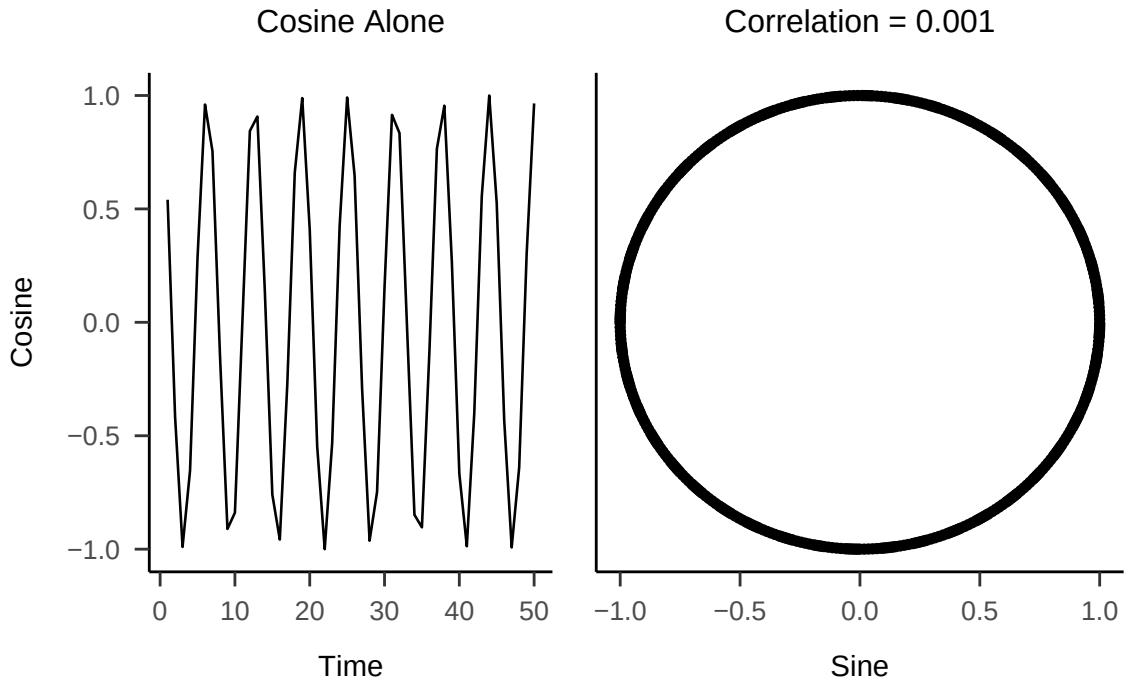
Data Visualization or data viz for short is an important part of reporting any data in any context. Drawing a pirate map where X marks the spot could be data visualization as well as a professional statistic of political vote tallies in bar plots. In general, humans are very visual creatures, so readers/ listeners can follow along more when there is something to look at that illustrates the point.



The type of visualization to choose strongly depends on what we want to accomplish, who the audience is and of course on the general setting. While the so-called *flametrees* above are nice to look at, the don't tell us too much about any data as-is. With reporting statistics, however, we want to be able to *see* our data ¯_( )_/¯. Having a visual representation can be helpful in understanding relationships between variables which helps us and consumers to interpret the results of analyses more intuitively. Moreover, many common statistics make assumptions about the *distribution* of our data, so we need to check and test that!

5.1.1 Example: Sine & Cosine

An illustrative example of why it can be very helpful to *see* data before making assumptions is the relationship between sine and cosine. We won't get too mathematical, but just know they are trigonometric angle functions with a special relationship. Commonly in psychology, relation = correlation, which assumes a linear relationship. However, when we take a look at both the correlation between sine and cosine as well as their relationship represented visually, we might see a problem:

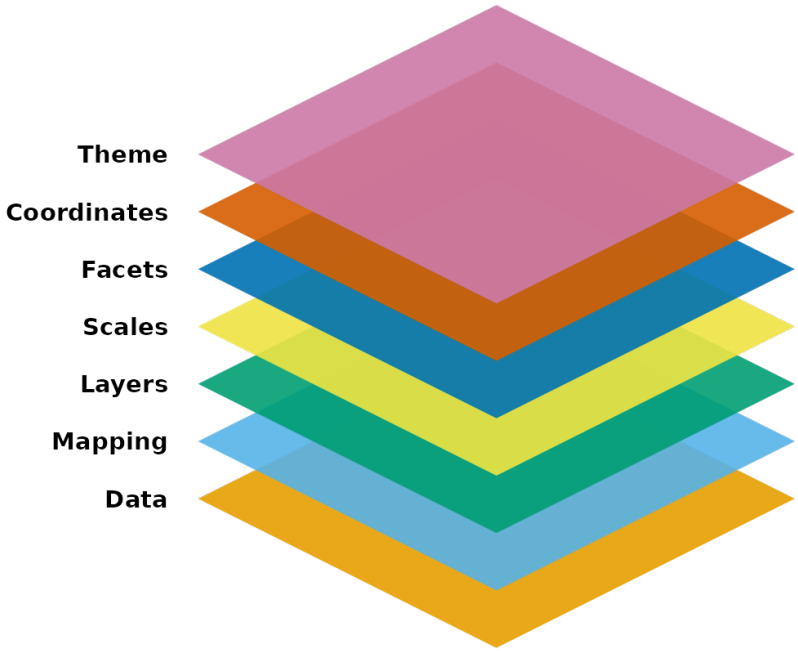


We can see that the two variables clearly have a certain type of relation with each other which is definitely not linear. But if we only reported the very very small correlation of .001, one could assume they had no relation at all! Keeping that in mind, let's see how we can create beautiful and concise visualizations in R!

5.2 What is ggplot2?

First off, it is probably the best data visualization package in R. Like `dplyr` it is part of the `tidyverse`, so you can expect fairly intuitively named functions and a clear structure that you can build up as you go. If you have not yet installed the whole `tidyverse`, you can install `ggplot2` individually with `install.packages("ggplot2")`. Once installed, you can load it with `library(ggplot2)`.

With `ggplot2` you can build your plots layer by layer. Here you can see a schematic of the different elements that can be added to any plot:



https://ggplot2.tidyverse.org/articles/ggplot2.html

There are two main functions, or rather types of function in this package.

`ggplot()`

This is the main function for "opening the canvas", so it basically prepares R for the plot definition. Commonly, we define the data set (the *Data* layer of the schematic) and which variables to plot in this function (the *Mapping* layer). The mapping needs the *aesthetics* function `aes()` as input, in which the variable(s) to be plotted will be defined.

> It's important to remember that the package is called *ggplot2* while the function call is *ggplot*!

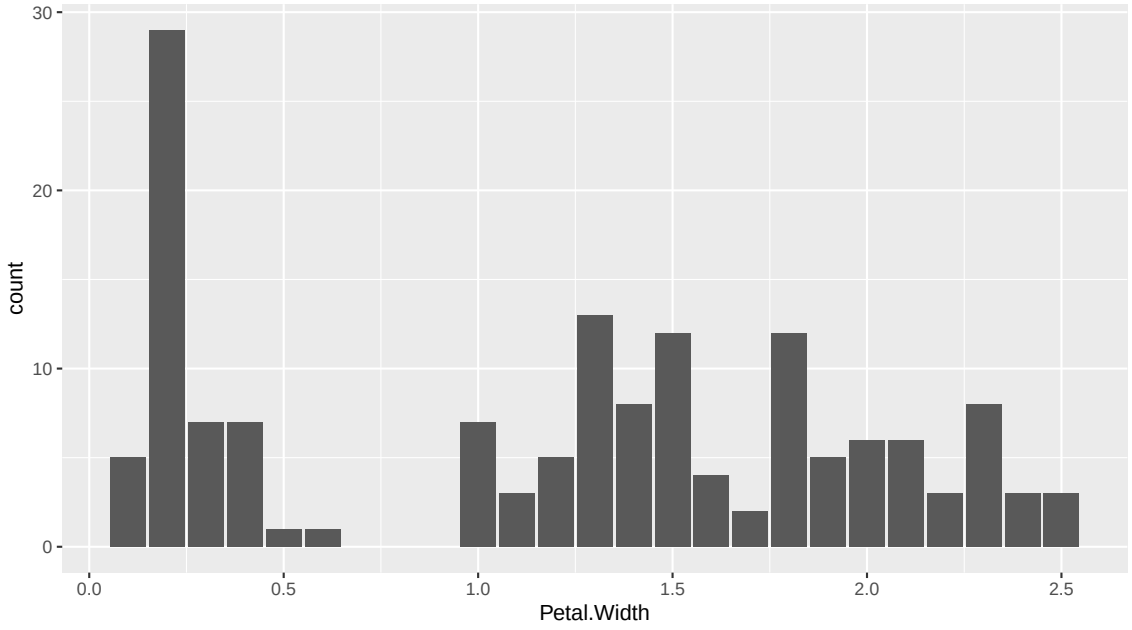
`geom_XYZ()`

- Defines the actual type of plot = *geometric objects*
- When data is pre-defined, this function does not *need* additional input

- Can handle some "pretty makers", such as `alpha`, which defines color opacity
- *geom_bar, geom_boxplot, geom_density, geom_jitter, geom_histogram*

5.3 Your First Bar Plot

```
ggplot(iris) +
  geom_bar(aes(x = Petal.Width))
```



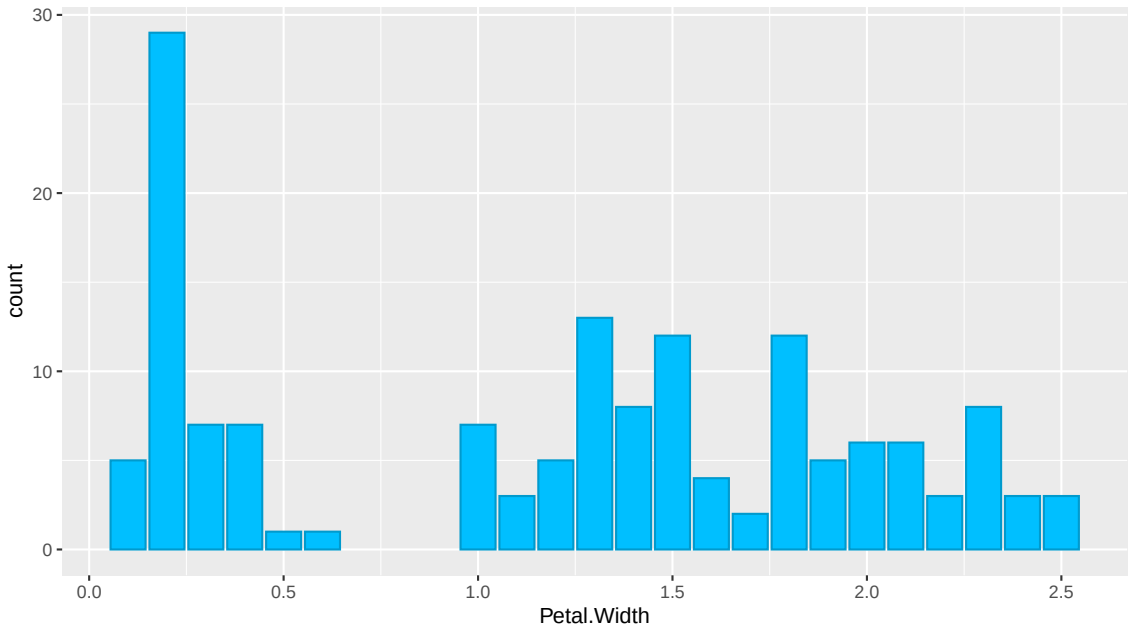
5.3.1 Making Plots Prettier

- color
  - visual property of the geometric object
  - which color for the outlines
  - `colors()`
- fill
  - visual property of the geometric object
  - which color to fill

- labs
- theme
  - `install.packages("papaja")` → `library(papaja)`

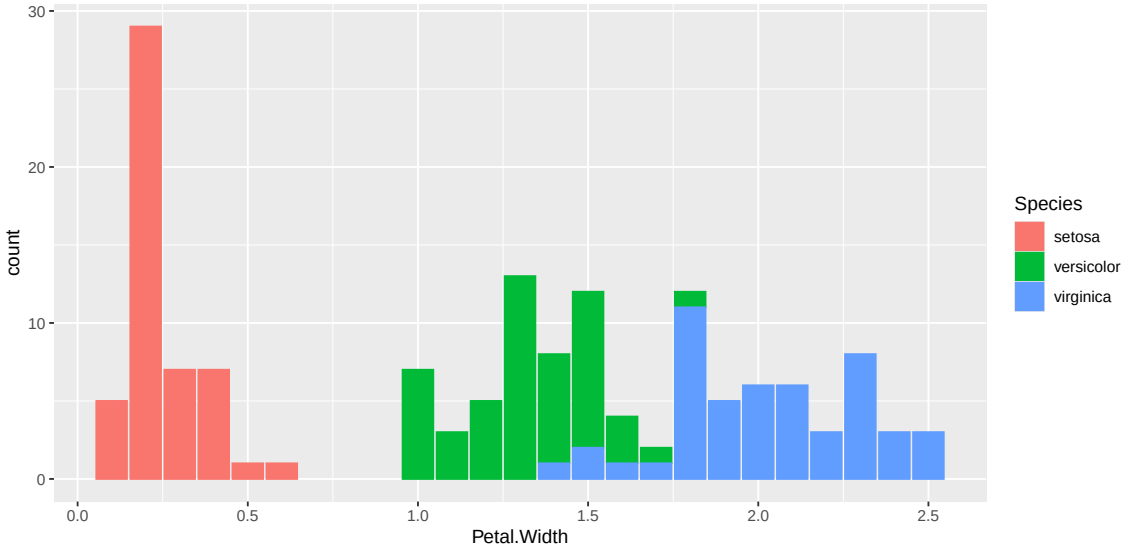
5.3.2 Example: Bar Plot with color and fill

```
ggplot(iris) +
  geom_bar(aes(x = Petal.Width), color = "deepskyblue3", fill = "deepskyblue")
```



What could cause this distribution?

```
ggplot(iris) +
  geom_bar(aes(x = Petal.Width, color = Species, fill = Species))
```



What do you notice about the code compared to before?

5.4 Static vs. Dynamic Aesthetics

- **Static Aesthetics**: Fixed values applied to all elements of the plot
  - Example: *color = "deepskyblue3"*
  - Means every element will have the same color.
- **Dynamic Aesthetics**: These map a variable in your data to an aesthetic, which allows different elements to have different colors based on the data
  - Example: *aes(color = Species)*
  - Means the color will vary according to the Species variable in the data set.

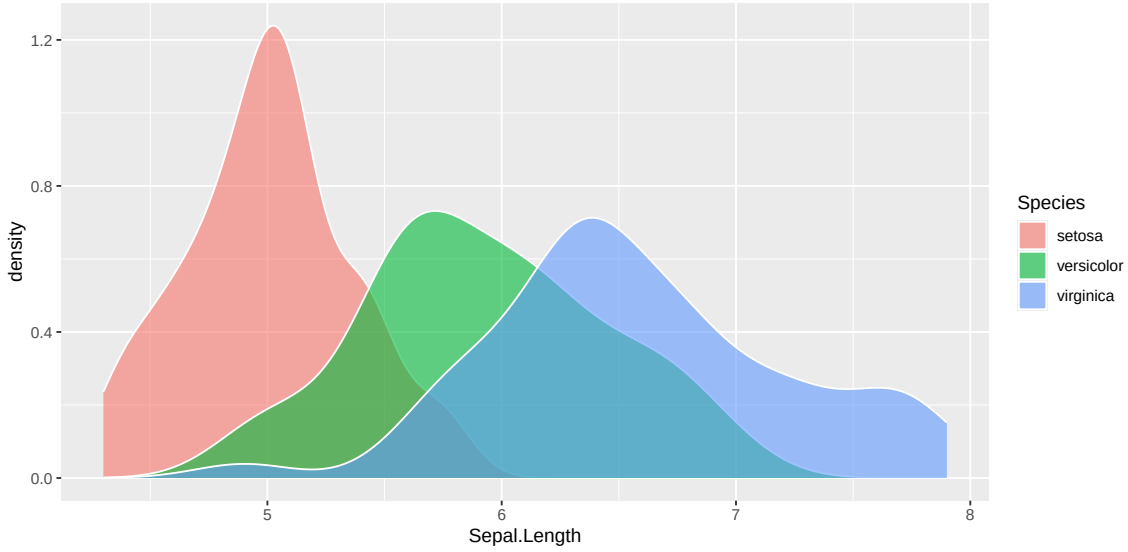
5.4.1 Exercise

Create a **density plot** that shows *Sepal.Length* from the `iris` data set. *Fill* in the color depending on the Species and *color* the outlines with "white". Make sure everything is visible and legible, so try to use an *alpha* of around 0.6.

```
ggplot(data) +
  geom_XYZ(aes(), alpha = ?)
```

5.4.2 Solution

```
ggplot(iris) +
  geom_density(aes(x = Sepal.Length, fill = Species),
               color = "white", alpha = .6)
```

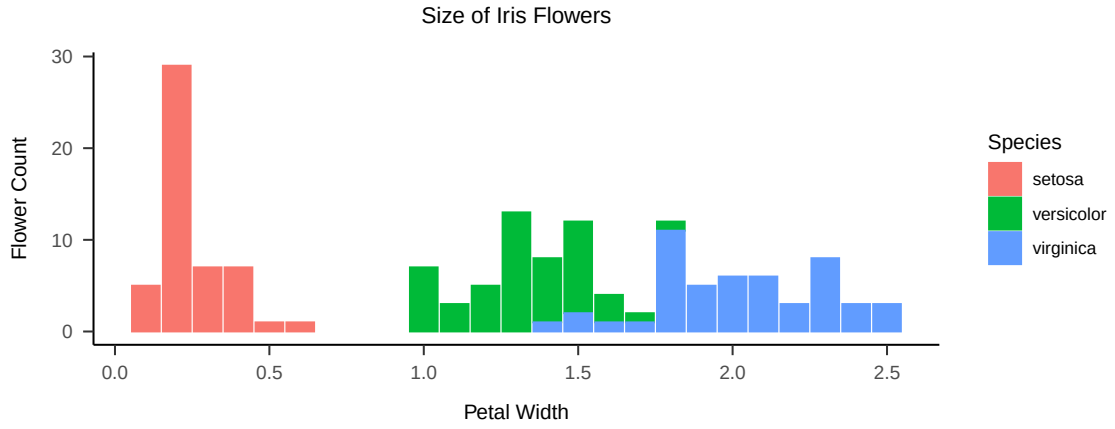


5.5 Adding labels and themes

- A good plot should be self explanatory and clear
- We need labels to tell others what our plot shows
  - Especially when using color for another variable, it needs to be clear what each color means
- Also the gray-ish default background is ok, but neither very pretty nor very clear
  - It is sometimes advisable to keep grid lines visible, but sometimes they can be distracting and unnecessary

  - ggplot2 has a lot of built-in theme options, but there are many packages that provide their own themes
  - Usually *my preference* is `papaja::theme_apa()`, which adheres to APA guidelines

```
ggplot(iris) +
  geom_bar(aes(x = Petal.Width, color = Species, fill = Species)) +
  theme_apa() +
  labs(x = "Petal Width", y = "Flower Count",
       title = "Size of Iris Flowers")
```



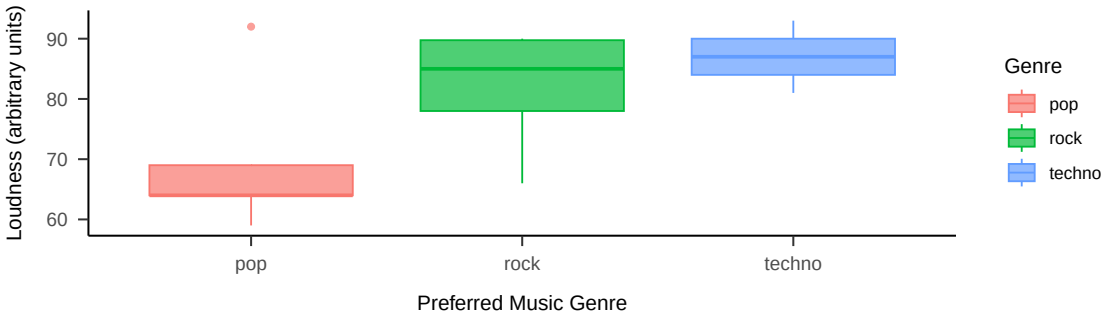
5.6 Bivariate Visualizations

- The bar plot shows the distribution of a single variable
  - We can add color to show groups
- Showing the relationship of two variables to each other is crucial for understanding our data
  - We can still add color to make existing groups clearer or add a third variable
- For a broad overview of which visualization (and statistic) to use for which type of data, visit the Statistics Picker (currently only available in German)
  - Most common are boxplots, scatterplots & line graphs

5.6.1 Boxplot Example

As suggested by one of you, we will look at the relationship of preferred music genre and music volume in our course:

```
# seminar <- readRDS("./data/seminar_data.Rds")
ggplot(seminar, aes(x=v07_genre, y=v08_loudness, colour=v07_genre, fill=v07_genre)) +
  geom_boxplot(alpha = 0.7) + theme_apa() +
  labs(x = "Preferred Music Genre", y = "Loudness (arbitrary units)",
       color = "Genre", fill = "Genre")
```

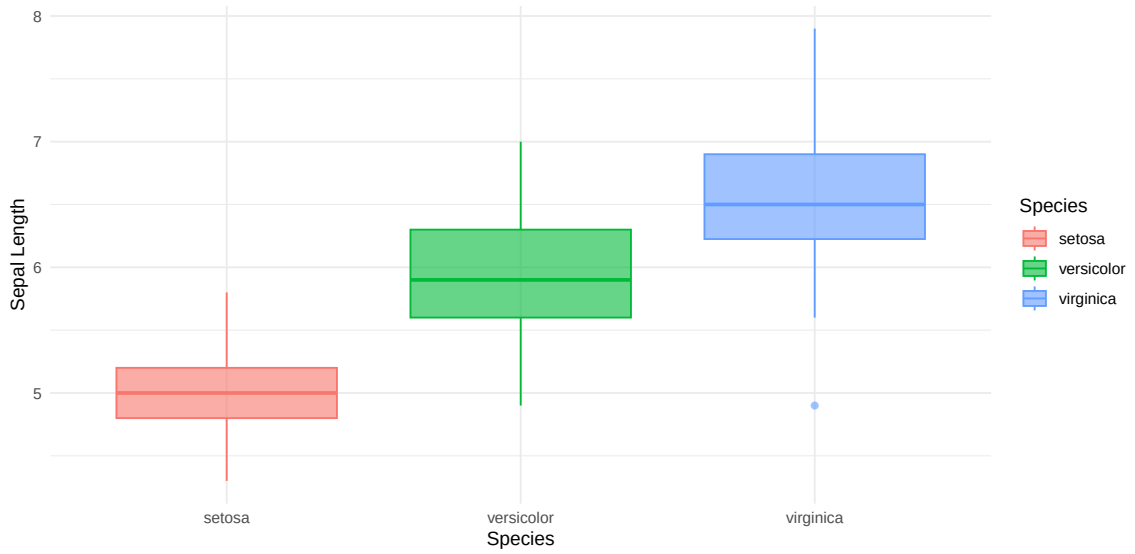


5.6.2 Exercise: Boxplot

Create a **box plot** that shows *Sepal.Length* from the `iris` data set grouped by the *Species*. *Fill* and *color* depending on the Species. Make sure everything is visible and legible, so try to use an *alpha* of around 0.7. Try to add *labels* and a *theme*.

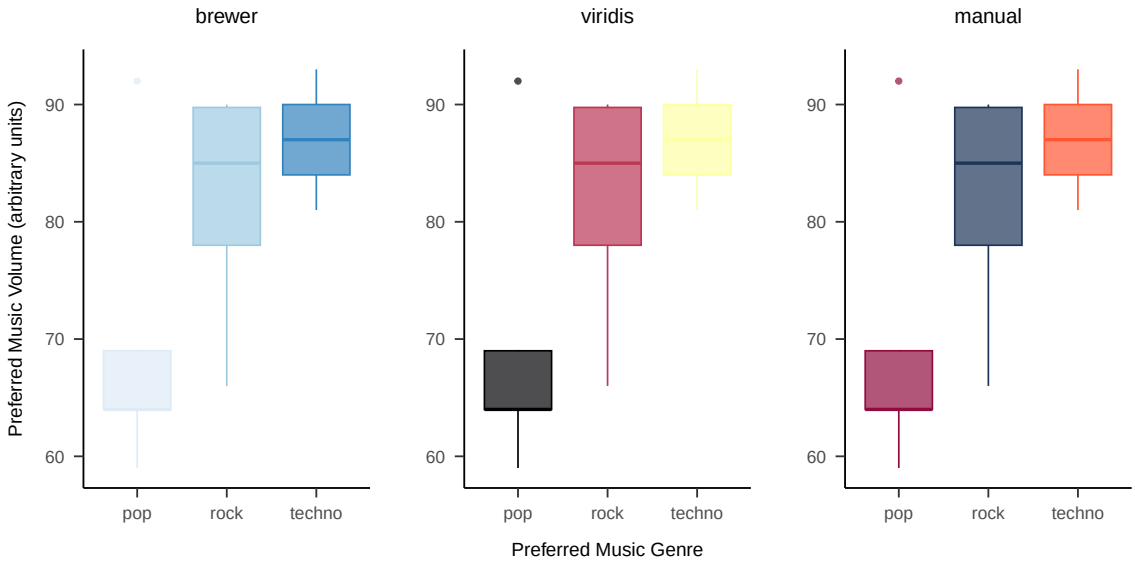
Solution

```
ggplot(iris) +
  geom_boxplot(aes(x = Species, y = Sepal.Length,
                   color = Species, fill = Species), alpha = .6) +
  theme_minimal() + labs(x = "Species", y = "Sepal Length")
```



5.6.3 Inspiration: colors and palettes

- These plots have all used the default colors from ggplot2
- There are many options for customization, either:
  - Use the "brewer" palettes from `ggplot2` with `scale_color_brewer()` or `scale_fill_brewer()`
  - Choose single colors (static aesthetics), check `colors()` for R color names
  - Create a color palette with all colors that you need and use it with `scale_color_manual()` or `scale_fill_manual()`
  - Use a predefined color palette from packages like `viridis` or `unikn` or `RColorBrewer`…
- Have fun with it!



Showing plots together like this is easy with `cowplot::plot_grid()`!

Wrap-Up & Further Resources

`ggplot2` is a powerful tool for visualizing data

Plot commands are added together with + and executed as one

A basic plot is created with ggplot() + geom_XYZ(), e.g. geom_bar

color & fill give you nice color options (static & dynamic)

labs() adds labels to the plot (i.e. x, y, title, …)

Themes control the background of the plot, e.g. papaja::theme_apa() or theme_minimal()

Color palettes are a great way of elevating a visualization

ggplot2 vignette

R Graphics Cookbook

unikn

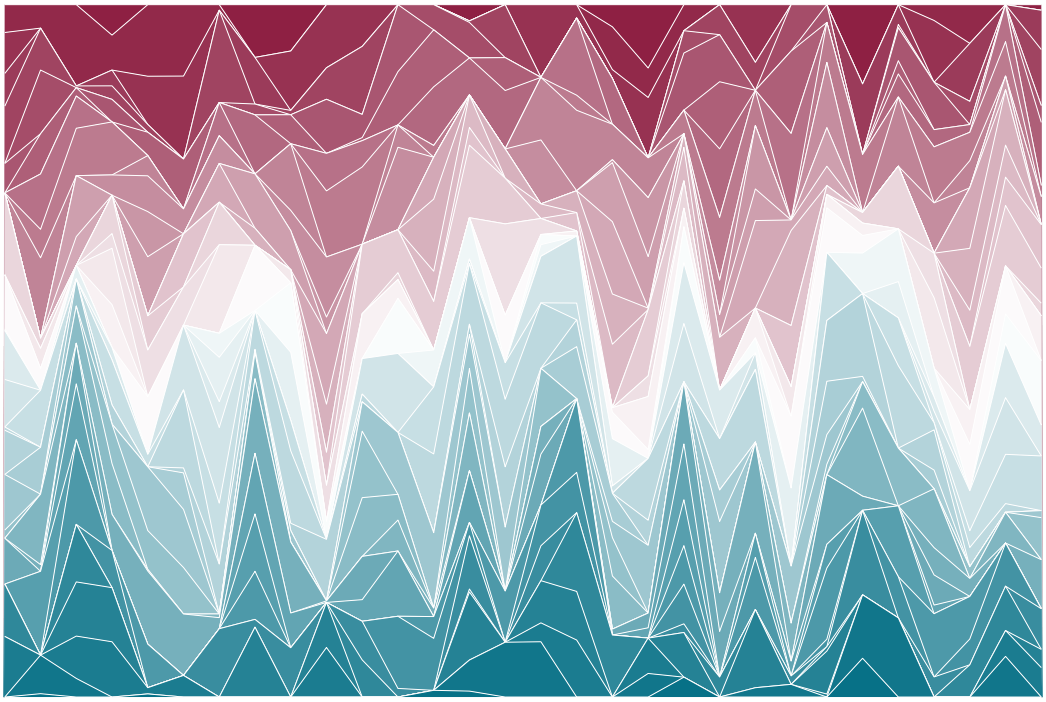
viridis

papaja

From Data to Viz Guide

Beautiful Plotting Guide Very extensive, I still profit from this guide a lot :)

More color palettes explained



Part II

The Nitty-Gritty

Chapter 6

Working with existing Data

After learning how to get your data to be the right format and how to look at said data, it now gets down to the nitty-gritty. In this chapter we will talk about where data can come from and how to get it inside of R. For that, we will talk about different ways of reading data into R to get the right format and also some first steps that should be taken with data in order to work with it properly.

We will work with some new packages, so please make sure you have the `readr` package as well as the `wobblynameR` package, which is only available on GitHub. To install it, you will first need the `devtools` package and then use it to install from GitHub:

```
install.packages("devtools")
devtools::install_github("the-tave/wobblynameR")
```

6.1 Different types of data sources

Data collected from WEXTOR - like from many other sources - will be exported in the CSV format, which stands for comma-separated values. That means that this type of data has one row per participant and the values for each column (variable) are separated by a *delimiter* [1].

[1] ...which often times is a comma, but not always! WEXTOR, for example uses semicolons.

When our data comes from an Excel file, it will often be exported as CSV because this type of storing data is pretty efficient and uses up very little space on the disk. A more modern but also more space-consuming way of storing Excel table data is in the XLSX format (Not an acronym, just stands for "Excel Spreadsheet").

Many people use SPSS for statistical analysis and creating dataframes. These data will be stored in a file with the extension SAV. Generally speaking, SAV files are compatible with R. However, there are some specialties which can cause problems, such as SPSS labels. We won't go into depth on this, just know that R can sometimes recognize SPSS labelled variables as their own class, in which case the class of those variables should be corrected manually [2].

Finally, of course we can also encounter datafiles that were created in R. In this case the will have .Rds as the file extension. A special case of this is when data frames are included in base-R or in a package. For example, the dataset called `iris` comes with R as an example dataset.

Therefore, we can run a function on it without having to load it at all:

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

Depending on which type of data we want to work with, there are different ways of loading the data into R.

6.2 Reading in the data

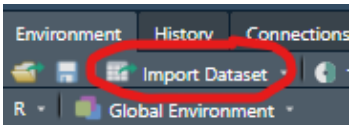
There are two main ways to load data: via the "Import Dataset" menu under the *Environment*-tab or (only) using code and if necessary packages. In the menu, simply select the option "Import

[2] The function calls `as.numeric()` and `as.factor()` will suffice in most cases.

Dataset" menu under the top-right *Environment*-tab and use *From text (readr)* for csv or other text files (such as .txt). Note that this menu also offers the possibility to load data from Excel and SPSS. However, this is much easier done by just using simple commands directly in your script:

Data Source	R command
WEXTOR (CSV)	`readr::read_delim("data.csv", delim = ";")`
Excel (XLSX)	`xlsx::read.xlsx("data.xlsx", sheetIndex = 1)`
R Data Source (RDS)	`readRDS("data.Rds")`
SPSS (SAV)	`foreign::read.spss("data.sav")`

You can follow along these exact code examples if you download the *seminar_data_raw* data frame from GitHub and save it in the data subfolder in your R-Project. You can find it under this link: https://github.com/the-tave/psych_research_in_r/raw/main/data/PsychResearchR_data.csv.[3]



Use the "Import Dataset" Menu under the *Environment* tab in the upper right corner of R Studio. In that menu, there are different options for different operations, but for CSV files I recommend the *From Text (readr)* option. Then we can select our data file and choose some basic aspects of reading in that data.

One of the reasons why i prefer this option over others is that a lot of the times it will recognize the data specifications on its own, so when you have selected the data file on you computer and copy out the generated Code from the Code Preview, it will look something like this:

Generated Code

```
library(readr)
seminar_data_raw <- read_delim("data/seminar_data_raw.csv")
```

[3] If you are connected to the internet anyway you can also use this link as the file path without downloading the data first!

```
View(seminar_data_raw)
```

The `read_delim()` function needs the file path to the data file as the minimum input requirement. There are many basic edits that can already easily be made in this function, which can also be selected in the menu. For example, even though "csv" stands for *comma-separated values* these types of files sometimes use a semicolon as a delimiter. When reading in the data, this is sometimes recognized automatically, but when the data looks a bit funky in the preview, making sure the right delimiter is selected is a solid first step.

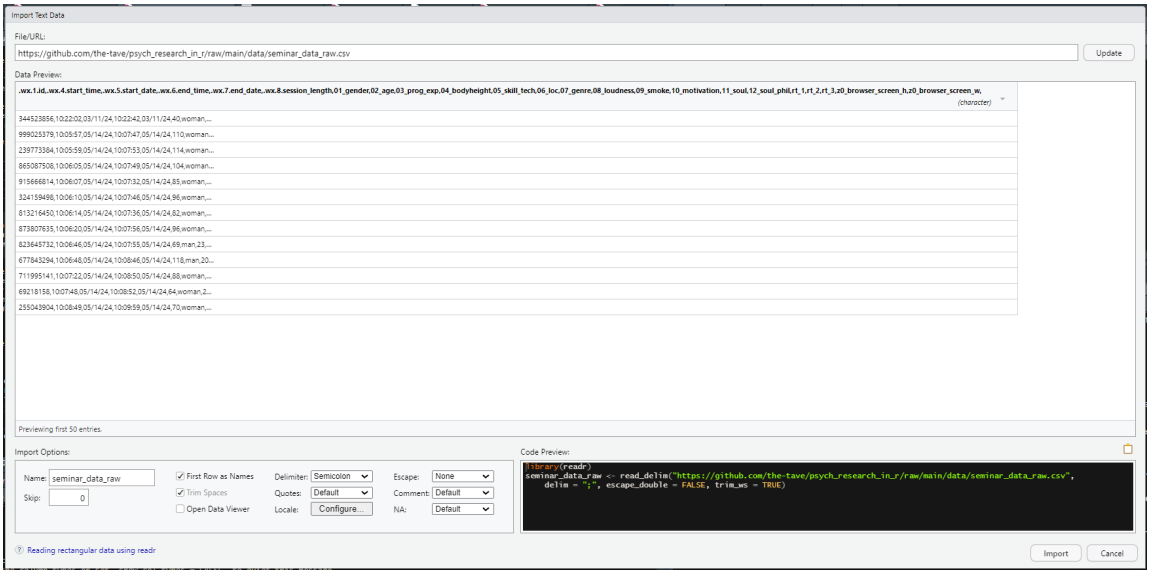


Figure 6.1: Example of wonky-looking data in the Import Dataset Menu

Basic Edits

There are a lot more quite basic edits that we can achieve from the comfort of the Import Dataset menu. In this case, we will make explicit some specifications that are automatically recognized, namely `delim = ","` - i.e. the correct delimiter, `escape_double = FALSE` - i.e. double quotes are not treated as a special character[4] and `trim_ws = TRUE` - i.e. if there are white spaces in the data, they should be trimmed.

[4]Escape characters are very important and very confusing. Just know that some special characters can be recognized either as plain text or as a request to do something (as code).

We can also specify some column types in this menu. Most columns will probably be fine staying the type they are automatically recognized as but especially some date formats are not properly recognized by R. So it makes sense to look through the data preview and do a quick visual check whether the data you see is displayed properly and the values make sense. If that check reveals that something is wrong, the little gray triangle next to the variable name offers many column types to choose from and find the one that suits your data. In our example, we will make sure that the start and end date variables are recognized as dates in the right format. For that we will choose the *Date* format and trigger a prompt asking for the correct format with some placeholders. When talking about dates, placeholders like m, d and y should be intuitive enough - suffice it to say that a capital Y stand for the whole year (e.g., 2024) while a small y stands for a shortened year format (e.g., 24).

Another handy column type is `col_skip()` which does just that - skip that specific column. It may seem strange to exclude some data from the get-go, but some data files will end with a delimiter and R recognizes that as another column. This is also the case for this data file from WEXTOR, which is why we will skip the last column (which is completely empty).

Moreover, we can take full advantage of both menu and script by copying the code that is generated from the menu into our script. This automatically gives us a nice structure to start a new script for analyses with a first library command at the top (for `readr`) and reading in the data as a first step of analysis. As a best practice, we will change the name of our raw data set in R to "raw". This data set can remain untouched throughout all other data munging steps which allows us to revise edits or go back a step if something goes wrong with having to reload the data (especially with a big data set, reading it in can take a bit of time).

As a last first step, we will use the `namepref0()` ("name-prefix-zero") function from the `wobblynameR` package to add the prefix "v" to every variable name. There are different reasons why a consistent prefix might be useful - in this case the variable names start with a number, which can cause problems in R. The function takes the whole data frame as input and outputs that data frame with all variables renamed, which we will save in a new data frame called "seminar". The resulting code - the first part of which can be completely generated by the menu - looks like this:

Result

```
# Library commands on top
library(readr)
library(wobblynameR)

# Generated code
raw <- read_delim("./data/seminar_data_raw.csv",
    delim = ",", escape_double = FALSE,
    col_types = cols(.wx.5.start_date = col_date(format = "%m/%d/%y"),
                     .wx.7.end_date = col_date(format = "%m/%d/%y"),
                     ...24 = col_skip()),
    trim_ws = TRUE)

# Add "v" as varname prefix to all variables
seminar <- namepref0(raw, "v")
```

6.3 Codebook

In my experience, it is easiest to work with data where you have an idea of what to expect. Even if you create your survey or experiment, you may forget what exactly is in which variable. With the `str()` function we can get a solid overview of the data structure and with `names(seminar)` R will output all variable names in the seminar data frame.

When you work with someone's data, it is a good idea to either consult their codebook. It might also help you in deciding what type of analyses might make sense with which variables.

6.4 Descriptives

Next to the general overview of what is contained in the data, descriptive statistics are very useful to look at for the variables of interest. Also, in any report, we need to give our readers or listeners an overview of the data to better judge the actual meaning of the results. Basically, we need to

know the sample to judge the results! A sample of e.g. 10 female psychology students probably shows different results - generally speaking - than a sample of 10 male soldiers.

Name some common descriptive statistics:

- _____
- _____
- _____
- _____
- _____

Show the solutions

- Mean
- Median
- Mode
- Variability, Variance, Standard Deviation
- Data Visualization…

6.4.1 group_by & summarize

With the `dplyr`-workflow, we can easily output group statistics with the keywords: `group_by()` and `summarize()` The code is built like any other `dplyr` workflow with pipes (%>%) in between each function. Commonly, you will first define the name of the data frame - in the first example we will use the `iris` data. Then choose the variable that should be used for grouping, such as gender or in this case *Species*. Make sure that this variable is recognized as either a factor or character and try to choose a variable that has not too many groups, otherwise it might get too crowded and confusing rather than helpful. Then you can pipe this grouped data into the summarize function and define some names for the measures that will we used in the output as well as the measure you would like to use - i.e. commonly `mean()` `median()` or `sd()`. You can also use `n()` to display the group sizes.

```
iris %>%
  group_by(Species) %>%
  summarize(m_plength = mean(Petal.Length),
            med_pwidth = median(Petal.Width),
            m_slength = mean(Sepal.Length),
            mode_swidth = getmode(Sepal.Width),
            n = n())
# # A tibble: 3 x 6
#   Species    m_plength med_pwidth m_slength mode_swidth     n
#   <fct>          <dbl>      <dbl>     <dbl>       <dbl> <int>
# 1 setosa          1.46        0.2      5.01         3.4    50
# 2 versicolor      4.26        1.3      5.94         3      50
# 3 virginica       5.55        2        6.59         3      50
```

6.5 Missing Data

In R, missing data or `NA`s (which stands for "Not Available") occur when some values in your dataset are absent or not recorded. This is important to note because missing data can affect your analyses and the conclusions you draw from your research. R treats these missing values as a special case, so they don't interfere with calculations in the same way as other values. You can identify missing data using functions like `is.na()`, and there are various methods to handle them, such as removing rows with missing values or using statistical techniques to estimate them.

This can easily be done with the `dplyr` tools we already know and a bit of logic: If a variable X in the data set *data* has one or more missing values, you can use

```
data_edited <- data |>
  filter(!is.na(X))
```

to filter in only those rows in the variable that as *not* NA (thus the !). Handling missing data properly ensures that your analysis is accurate and reliable.

Exercises

group_by & summarize

Follow the structure to group the edited seminar data by belief in the soul and output a summary with

- mean age
- mean technological skill
- mode music genre
- median volume of music
- grouped n (function *n()*)

Solution

```
seminar %>%
  group_by(v11_soul) %>%
  summarize(m_age = mean(v02_age),
            m_tech_skill = mean(v05_skill_tech),
            mode_music = getmode(v07_genre),
            median_volume = median(v08_loudness),
            n = n())
# # A tibble: 3 x 6
#   v11_soul m_age m_tech_skill mode_music median_volume     n
#   <chr>    <dbl>        <dbl> <chr>              <dbl> <int>
# 1 dunno     23.4         29.4 techno              89       5
# 2 no        22           45   pop                 74.5     2
# 3 yes       22.5         42   rock                67.5     6
```

Prepare seminar data

We have read in the "seminar_data_raw" and looked at some of the basic edits that we can do easily. However, in order to be able to work with the dataset in easily in the future, we should do some more advanced edits. Specifically, we will:

1. Change the date format so we get a start and end variable including time and date instead of separate date and time variables
2. Remove the unnecessary original date and time variables
3. Change the one missing value in `v12_soul_phil` to "dunno"
4. Rename the ID variable to `ID` & session length variable to `session_length`
5. Make our data type explicit using `as.data.frame()`
6. Save our edited data in an R data set (`Rds`) format so we can use it more easily later!

I suggest using some of the `dplyr` commands that were introduced in the last chapter for edits 1 through 4. You can try to figure them out on your own and later look at the solution below. **I highly recommend you follow along with these steps, as we will use this dataset in later chapters**[5]

Solution

```
seminar <- seminar |>
  # 1. Change date format
  mutate(start = as.POSIXct(paste(v.wx.5.start_date, v.wx.4.start_time), format = "%Y-%m-%d %H:%
         end = as.POSIXct(paste(v.wx.7.end_date, v.wx.6.end_time), format = "%Y-%m-%d %H:%M:%S")
         .before = v01_gender) %>% # it is more efficient to have date and time together
  # 2. Remove (now) unnecessary original date vars
  select(-c(v.wx.4.start_time, v.wx.5.start_date, v.wx.6.end_time, v.wx.7.end_date)) %>%
  # 3. Change missing value to "don't know" - in this case makes sense in context
  mutate(v12_soul_phil = ifelse(is.na(v12_soul_phil), "dunno", v12_soul_phil)) |>
  # 4. Rename the ID & session length variable
  rename(ID = v.wx.1.id,
         session_length = v.wx.8.session_length)

# 5. Make data type explicit
seminar <- as.data.frame(seminar)
```

[5]This is very close to real-life practice, but you can also skip it and download the Rds-file from the GitHub repository here.

```r
# 6. Save edited data frame in Rds format
saveRDS(seminar, "./data/seminar_data.Rds")
```

Wrap-Up & Further Resources

Reading in data works with commands like readRDS() but also the “Import Dataset” Menu

Use the menu for WEXTOR data to get an overview and reproducible code

Get/ create a codebook for your data

Always report descriptive data

dplyr’s `group_by()` and `summarize()` can output grouped descriptives

Reading in Data in Different Formats

Stats and R: Descriptives

Chapter 7

Loops and Conditionals

7.1 What is a loop?

- Loops are automations that take care of repetitive tasks
  - Typically, we might go through data row by row and perform a task
- Loops can have different forms:
  - **for**: loop over a pre-defined set of values, such as a number sequence
  - **while**: define constraints and keep loop running as long as they are met
  - **repeat**: loop repeats until broken (*I do not recommend this ever!*)
- For is most common, while is nice for more advanced simulations

7.1.1 What are conditionals?

- Conditionals define different actions for different conditions
- They follow intuitive language:
  - **if**: define action to take when condition is met (logical)
  - **else if**: define action for another condition
  - **else**: define action for everything else
- **ifelse** is the shortcut-function we already got to know for recoding with `mutate()`
  - defines exactly 1 condition, 1 action, 1 alternative

  - expandable version of ifelse is case_when

7.2 Exercise: What should I use?

1. I want to print numbers from 1 to 7!
   - for loop (but realistically *1:7* ;))
2. I need to simulate more data as long as my sample size is smaller than 200!
   - while loop
3. I want to print the age of only the females in my sample!
   - if/ else

7.3 for-loop

- In a for loop, we need to define the range *for* which the loop should run
- We define an *iterating variable* that will change each time the loop runs and take on all values of the defined range
  - Typically, we call this variable "i"
- We define the range in round brackets and all actions the loop should take in curly brackets

7.3.1 Example

```
for(i in 1:4){
  print(paste("The iteration is", i))
}
# [1] "The iteration is 1"
# [1] "The iteration is 2"
# [1] "The iteration is 3"
# [1] "The iteration is 4"
```

7.3.2 Conditionals: If - else

```
if(variable == "value"){
  print("Do something")
} else if(variable == "other_value"){
  print("Do some other thing")
} else {
  print("Do anything else")
}
```

7.3.3 Conditionals: If - else

7.3.3.1 Checking numbers |

We want to test whether numbers in a loop are divisible by 3. If they are, we will display "Divisible by 3", and if not, we will simply output the iteration number like before.

For that we will use the modulo %% , which gives us the rest of a division. Let's try it out!

```
for(i in 1:4){
  if(i %% 3 == 0){
    print("Divisible by 3!")
  } else {
    print(i)
  }
}
# [1] 1
# [1] 2
# [1] "Divisible by 3!"
# [1] 4
```

7.3.4 Exercise

Use a for-loop to display your favorite numbers. The numbers should be between 10 and 25.

If the number is divisible completely by 5, you should output “Divisible by 5!”, if not please output the iteration number only.

Solution

```
for(i in 10:25){
  if(i %% 5 == 0){
    print("Divisible by 5!")
  } else {
    print(i)
  }
}
# [1] "Divisible by 5!"
# [1] 11
# [1] 12
# [1] 13
# [1] 14
# [1] "Divisible by 5!"
# [1] 16
# [1] 17
# [1] 18
# [1] 19
# [1] "Divisible by 5!"
# [1] 21
# [1] 22
# [1] 23
# [1] 24
# [1] "Divisible by 5!"
```

7.4 Same function, different variables

```
for(i in c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")){
  print(mean(iris[ , i]))
}
# [1] 5.843333
# [1] 3.057333
# [1] 3.758
# [1] 1.199333
```

Brainteaser : What is the class() of i here?

character

7.4.1 More sophisticated

```
vector <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")

for(i in 1:length(vector)){
  print(paste("The mean value of", vector[i], "is",
              round(mean(iris[ , vector[i]]), 2)))
}
# [1] "The mean value of Sepal.Length is 5.84"
# [1] "The mean value of Sepal.Width is 3.06"
# [1] "The mean value of Petal.Length is 3.76"
# [1] "The mean value of Petal.Width is 1.2"
```

Brainteaser : And what is the class() of i now?

integer (numeric)

7.4.2 With saving values

```
vector <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")
means <- rep(NA, length(vector)) # create "empty" vector to fill

for(i in 1:length(vector)){
  print(paste("The mean value of", vector[i], "is",
              round(mean(iris[ , vector[i]]), 2)))
  means[i] <- round(mean(iris[ , vector[i]]), 2)
}
# [1] "The mean value of Sepal.Length is 5.84"
# [1] "The mean value of Sepal.Width is 3.06"
# [1] "The mean value of Petal.Length is 3.76"
# [1] "The mean value of Petal.Width is 1.2"
```

7.4.3 Exercise: Seminar Data

Please download an edited version of our seminar data from ILIAS. You will find it under *data* as a zip folder. The dataset in it should have ".Rds" as a file ending. Read it in using:

```
seminar <- readRDS("./data/seminar_data.Rds")
```

Create a for-loop that saves the means of age, body height and technological skill from that seminar data in a vector named "seminar_means".

Hint: I renamed the variables to make them nicer to work with, so check out names(seminar)!

7.4.4 Solution

```
vector <- c("v02_age", "v04_bodyheight", "v05_skill_tech")
seminar_means <- rep(NA, length(vector)) # create "empty" vector to fill

for(i in 1:length(vector)){
  seminar_means[i] <- round(mean(seminar[ , vector[i]]), 2)
```

```
}
```

7.4.5 Checking What's Going On

7.4.5.1 browser()

- In order to break our loop and enter a "debugging" mode, we can add browser() to any loop like so:

```
vector <- c("v02_age", "v04_bodyheight", "v05_skill_tech")
seminar_means <- rep(NA, length(vector)) # create "empty" vector to fill


for(i in 1:length(vector)){
  seminar_means[i] <- round(mean(seminar[ , vector[i]]), 2)
  browser()
}
```

- The loop will run the first iteration until browser is called and then pause
- It allows you to check in the console whether all variables look the way you want them to without having to wait for the loop to run
  - Or if there are errors/ warnings, you can check *when* they occur by moving the browser() to a different location in the code
- To continue with the next iteration while in browser, type "c" into the console, "Q" to quit
  - The Esc key will also terminate the browser
- **Make sure you exit it** before continuing with anything else

7.5 Loop with Conditional

In our seminar data, the **average body height is 167.23 cm**. We want to create a loop that goes through all rows (all seminar students) and outputs their height and whether it is above or below average.

What do we need for that?

7.5.1 Elements

- For each person:
- Check *if* their body height is above average
  - If yes: Output "This person is taller than average with …"
  - Else: Output "This person is less tall than average with…"
- Then paste the person's body height
- Loop that over each person!

Solution

```
for(i in 1:nrow(seminar)){
  if(seminar$v04_bodyheight[i] > mean(seminar$v04_bodyheight)) {
    print(paste("This person is taller than average with", seminar$v04_bodyheight[i], "cm."))
  } else {
    print(paste("This person is less tall than average with", seminar$v04_bodyheight[i], "cm."))
  }
}
# [1] "This person is taller than average with 168 cm."
# [1] "This person is less tall than average with 160 cm."
# [1] "This person is taller than average with 171 cm."
# [1] "This person is less tall than average with 166 cm."
# [1] "This person is taller than average with 169 cm."
# [1] "This person is less tall than average with 164 cm."
# [1] "This person is less tall than average with 161 cm."
# [1] "This person is less tall than average with 166 cm."
# [1] "This person is taller than average with 183 cm."
# [1] "This person is taller than average with 172 cm."
# [1] "This person is less tall than average with 160 cm."
# [1] "This person is less tall than average with 164 cm."
# [1] "This person is taller than average with 170 cm."
```

7.5.2 And a more complicated outlook...

7.5.2.1 What happens here?

```
for(i in 1:nrow(seminar)){
  if(seminar$v11_soul[i] == "yes"){ # Check soul belief
    text1 <- "This person belives in the soul and"
  } else if (seminar$v11_soul[i] == "no") {
    text1 <- "This person does not belive in the soul and"
  } else { text1 <- "This person is unsure about the soul and" }

  if(seminar$v12_soul_phil[i] == "monism"){  # Check mind-body-philosophy
    text2 <- "believes the m-b-relationship is monistic."
  } else if (seminar$v12_soul_phil[i] == "dualism") {
    text2 <- "believes the m-b-relationship is dualistic."
  } else { text2 <- "is unsure of the m-b-relationship." }

  print(paste(text1, text2)) # Print text
}
```

7.5.3 Result

```
# [1] "This person belives in the soul and believes the m-b-relationship is dualistic."
# [1] "This person belives in the soul and is unsure of the m-b-relationship."
# [1] "This person belives in the soul and believes the m-b-relationship is monistic."
# [1] "This person is unsure about the soul and is unsure of the m-b-relationship."
# [1] "This person belives in the soul and is unsure of the m-b-relationship."
# [1] "This person does not belive in the soul and believes the m-b-relationship is monistic."
# [1] "This person is unsure about the soul and is unsure of the m-b-relationship."
# [1] "This person is unsure about the soul and is unsure of the m-b-relationship."
# [1] "This person does not belive in the soul and is unsure of the m-b-relationship."
# [1] "This person belives in the soul and believes the m-b-relationship is monistic."
```

```
# [1] "This person is unsure about the soul and believes the m-b-relationship is dualistic."
# [1] "This person belives in the soul and believes the m-b-relationship is dualistic."
# [1] "This person is unsure about the soul and is unsure of the m-b-relationship."
```

Wrap-Up & Further Resources

For-Loops must have a defined sequence to run over, e.g. 1 to 4

For-loops are the most common in R and are fit for most tasks

Loops are a powerful tool to e.g. easily perform the same task many times

Conditionals give you control over different operations depending on your data

In combination, loops and conditionals are most useful

Don't forget to play and create fun loops!

For loops (W3 Schools)

For loops (R bloggers)

For loops (DataMentor) Here are also links the the other types of loop and a visualization

Loops in R (datacamp) Very thorough! :)

dplyr Exercises Great to keep practicing over the holidays :)

Lightbot

https://medium.com/nerd-for-tech/what-are-for-loops-b7215db28e83

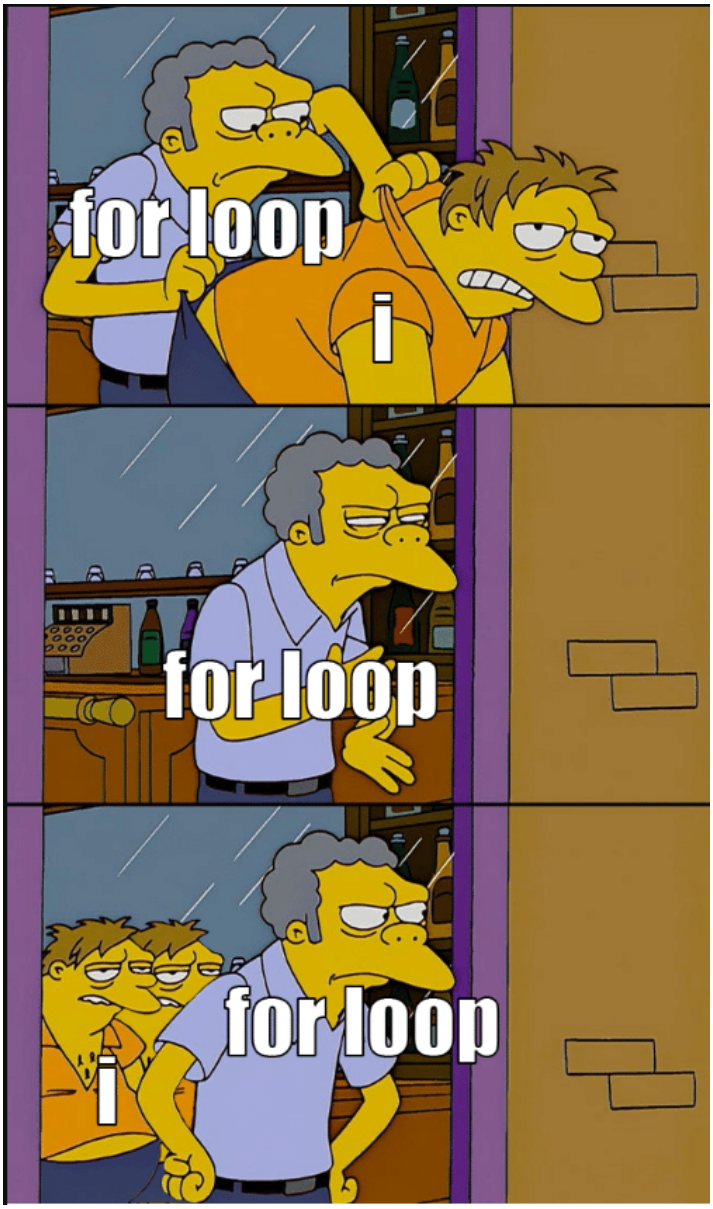


Figure 7.1: For Loop Meme

Chapter 8

Difference Statistics

In this chapter, you will learn how to conduct some of the most common statistical analyses in R. We will start with a re-cap of the statistics to get everyone on the same page, going over why the χ^2 test, t-test and ANOVA are all in the same chapter and when to use each. Then I want to mention some assumptions we have to check and lastly go over some examples fueled by students' ideas who attended my seminar that this work is based on.

8.1 Statistics Re-cap

Think about what do you remember from statistics and try to answer the two questions below:

- What is Chi^2 / χ^2?
- What is the t-test?
- What is an ANOVA?

What do they have in common?

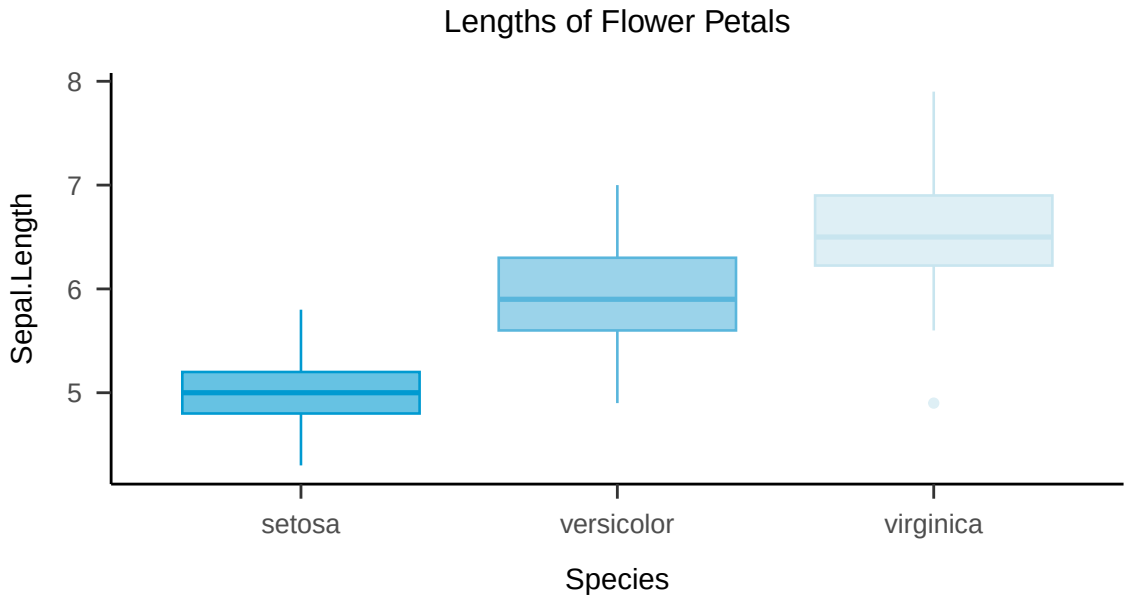
Solution

They measure *group differences*

What differentiates them?

Solution

They are used for data of different scale levels



These three measures all provide us information on *differences* either between naturally occurring groups (e.g. people of different genders) or between experimentally manipulated sub-samples (e.g. treatment vs. control group in a medical drug trial).[1]

The χ^2 statistic is commonly used on nominal data, e.g. if you want to compare proportions. Usually, the χ^2 test is used with two binary variables, but it also works with more categories. Considering the setup with the drug trials, we could check whether women and men received the drug and the placebo equally often.

With the t-Test it gets a bit trickier since there is more than one variant: Depending on the situation you can use the one-sample, two-sample or paired-samples t-test. Generally speaking, you want to test one or two groups on a continuous attribute.

In the analysis of variance $\rightarrow$ *ANOVA* you want to test three or more groups with one (or more)

[1] You will usually see the Chi^2 test classified as an association statistic, which is technically more to the point. However, I like to think of it as quantifying differences between groups. After all, if you find systematic differences between groups in two given measures, that must mean that those measures are not independent, i.e. associated with each other.

continuous attribute(s). I like to think of it as an augmentation of the t-test which takes into account some issues that we would face if we just used several t-tests in a setting with more than two groups (spoiler: It's called alpha-inflation and we do not want it).

This is a very basic overview of these tests, but it should give you a reminder if you have heard this before. I will go over the purposes again as we look at the coding examples below. If you think you might need a refresher on these stats that goes a little deeper, hop on over to the Stats Picker and to the Deep Dive tab. There you will find some more information on multivariate statistics and how to use them.

8.1.1 Pre-Requisites

If you have been following along with the structure of this book, you will know what an R project is and that you were asked to create a folder called "data" in the according project folder in order to make the following command work to load the seminar data. If you just want to access the tidily cleaned data directly, you can go to the GitHub page and download the datafile there. Alternatively, there is a direct way to include data from GitHub in your script as long as you are connected to the internet, which you can find commented out below:

```
seminar <- readRDS("./data/seminar_data.Rds")


# Access the data directly from GitHub
# seminar <- readRDS(gzcon(url("https://github.com/the-tave/psych_research_in_r/raw/main/data/seminar_d
```

Next, we will add a so-called *dummy-variable* about believeing in the soul. It is not called "dummy" because we want to judge anybody's belief, but rather because it serves as a stand-in for a more complex variable, just like a test dummy is a stand-in for a real person.[2]

Here, we basically want to create a dummy variable for believing in the soul where we summarize all those who said they don't know or they definitely don't believe into one category that we can later contrast with all those who said they do believe. So in the new variable, 1 means *yes* and 0 means *no or unsure*.

[2]For more in-depth information on dummy variables you can visit: https://stats.oarc.ucla.edu/wp-content/uploads/2016/02/p046.pdf.

```
seminar$soul_dummy <- ifelse(seminar$v11_soul == "yes", 1, 0)
```

8.1.2 Statistical Significnace

In parametric statistics (which is the most common/basic kind), we make our test decision based on the distribution of out given test statistic. This is because for any test statistic that we can calculate, we can make a prediction about its **distribution** if the true value were zero, i.e. no effect. We would expected most values to fall in a certain range and if we find an empirical value (based on our data) that falls beyond that range, we conclude that it is so unlikely to find a value that extreme if we actually assume that range to be surrounding the true value. We then conclude statistical significance because the empirical value of the test statistic is significantly different to the theoretical value distribution we would have if there were no true effect.

This measure of "how unlikely is our empirical value" is reflected in the **p-value**. Commonly, we accept a threshold of $\alpha = .05$ as sufficiently unlikely. It means that we would theoretically expect a value that extreme or more extreme in only 5% of all cases.

8.2 χ^2

The χ^2 Test for Independence determines if there is an association between **two categorical variables** in a contingency table. To do so, it compares the observed frequencies in each category of that contingency table to the frequencies expected if the variables were independent. For example, if there were no association (i.e. *statistical independence*) between gender and believing in the soul, then the proportion of men and women who believe in a soul should be about equal.

```
#
#           0  1 Sum
#   man     1  1   2
#   woman   6  5  11
#   Sum     7  6  13
```

The test statistic χ^2 is calculated as follows:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

where - O_i is the observed frequency in each category - E_i is the expected frequency in each category, calculated as:

$$E_i = \frac{(row\ total) \times (column\ total)}{grand\ total}$$

The distribution of χ^2 depends on the number of categories k or more precisely, the degrees of freedom, df. In a scenario where we have just one variable and we want to test empirical frequencies to expected ones, k is equal to the number of categories and df = k-1. If we have two variables and we want to test their association as described above, k is not really relevant here and we focus on the degrees of freedom, calculated as df = (#rows - 1)x(#columns - 1).

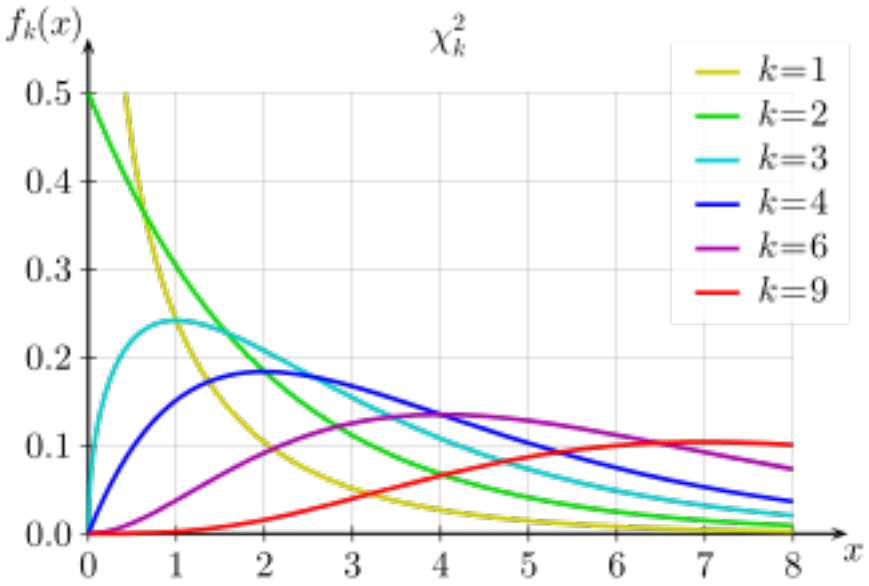


Figure 8.1: Chi squared distribution

8.2.1 How to in R

In R, you calculated the Chi^2 test using the function `chisq.test()`, which only needs (categorical!) *x, y* as input. Here, we will look at the previous example of gender and soul belief and therefore add those two variables. The function allows some other inputs as you can see in the example.

Especially with a small sample, we can add the input *simulate.p.value* = *T*, which bootstraps the analysis 2000 times to better estimate an accurate p-value.

```
chisq.test(seminar$v01_gender, seminar$soul_dummy, simulate.p.value = T)
#
#   Pearson's Chi-squared test with simulated p-value (based on 2000
#   replicates)
#
# data:  seminar$v01_gender and seminar$soul_dummy
# X-squared = 0.014069, df = NA, p-value = 1
```

8.2.2 Exercise

We want to explore whether **belief in the soul (dummy)** is associated with **music preference**.

Calculate a simple chisq.test and interpret the results.

Solution

```
chisq.test(seminar$v07_genre, seminar$soul_dummy)
# Warning in chisq.test(seminar$v07_genre, seminar$soul_dummy): Chi-squared
# approximation may be incorrect
#
#   Pearson's Chi-squared test
#
# data:  seminar$v07_genre and seminar$soul_dummy
# X-squared = 2.1357, df = 2, p-value = 0.3437
addmargins(table(seminar$v07_genre, seminar$soul_dummy))
#
#           0  1 Sum
#   pop     2  3   5
#   rock    3  3   6
#   techno  2  0   2
#   Sum     7  6  13
```

How would you interpret this result?

- (A) Dualists prefer Rock music.
- (B) Music preferences influence the soul beliefs of a person.
- (C) There is no significant association between belief in the soul and music preference by genre.
- (D) Chi squared is confusing.

8.3 t-Test

As I've mentioned before, there is not really *the one* t-test to rule them all. Instead, the t-statistic which the t-tests are built around is calculated slightly differently depending on the actual situation. There are three categories of t-test:

- **one-sample** t-test:
  - Test one sample against a known mean value
- **two-sample** t-test (independent):
  - Test two sample-means against each other (independent samples)
- **paired two-sample** t-test:
  - Test two dependent sample-means against each other (e.g. repeated measures)

8.3.1 Test statistic *t*

The Test statistic T has a known distribution that depends on the **degrees of freedom df**, calculated as n-1. It is similar to what we saw with the χ^2 distribution but the t-distribution looks a lot more like the normal distribution. Most (probable or to be expected) T values are around 0, so te further away or *more extreme* the T value is from 0, the less likely it is caused by chance alone. $\rightarrow$ *significance*

It is calculated as follows:

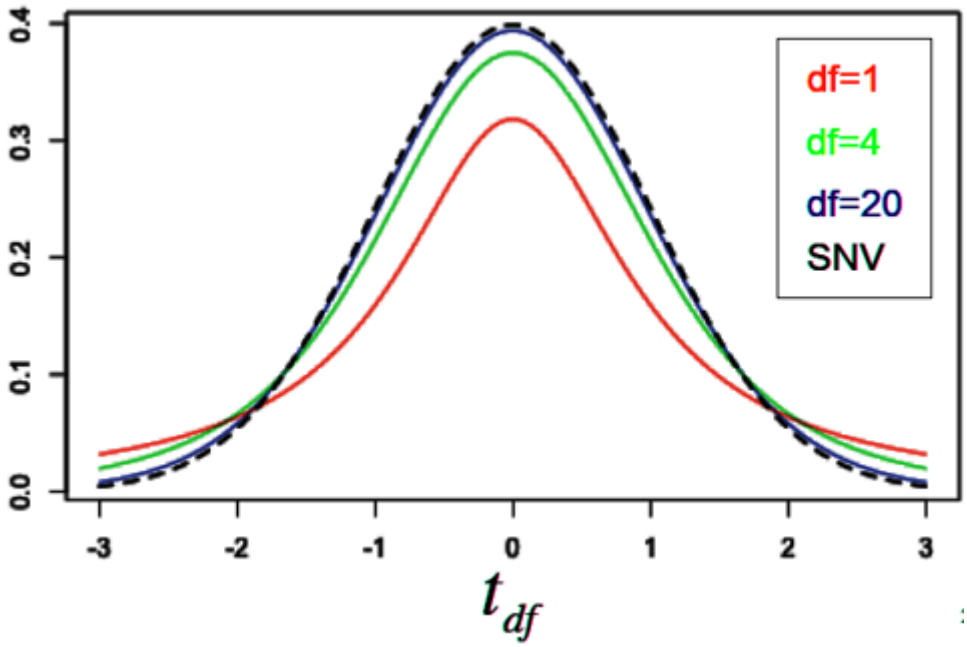


Figure 8.2: T distribution

- One sample:

$$t = \frac{\bar{X} - \mu}{\frac{s}{\sqrt{n}}}$$

- Two sample:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

8.3.2 How to in R

- The basic function is `t.test()` for any type of t.test
- One sample needs inputs *x, mu* (if x is from a data set you should specify *data*)
- Two sample needs either
  - *x, y* (if from data set, *data*) or
  - *x* ~ *group* (if from data set, *data*)
- Paired test needs *paired* = *T*
- One-sided test needs *alternative* = *'greater'* (assumes first group mean to be larger than

second; otherwise “less”)

8.3.3 Examples

One sided - “greater” assumes that mean(x) is larger than mean(y)

```
t.test(x = 10:20, y = 0:10, alternative = "greater")
#
#   Welch Two Sample t-test
#
# data:  10:20 and 0:10
# t = 7.0711, df = 20, p-value = 3.713e-07
# alternative hypothesis: true difference in means is greater than 0
# 95 percent confidence interval:
#  7.56088     Inf
# sample estimates:
# mean of x mean of y
#        15         5
```

Two sided using formula notation with the ~ tilde

Does seminar motivation differ depending on the soul-belief of students?

```
t.test(v10_motivation ~ soul_dummy, data = seminar)
#
#   Welch Two Sample t-test
#
# data:  v10_motivation by soul_dummy
# t = 0.1625, df = 6.833, p-value = 0.8756
# alternative hypothesis: true difference in means between group 0 and group 1 is not equal to 0
# 95 percent confidence interval:
#  -31.14063  35.71206
# sample estimates:
# mean in group 0 mean in group 1
```

```
#        68.28571        66.00000
```

8.3.4 Exercise

We want to test whether the **gender** stereotype that men are **more skilled with technology** appears in our seminar sample.

Perform a one-sided two-sample t-test and interpret the results.

Hint: The grouping variable "v01_gender" is sorted alphabetically - so choose the "alternative" accordingly!

Solution

```
t.test(v05_skill_tech ~ v01_gender, data = seminar, alternative = "greater")
#
#   Welch Two Sample t-test
#
# data:  v05_skill_tech by v01_gender
# t = 0.38715, df = 1.2774, p-value = 0.3766
# alternative hypothesis: true difference in means between group man and group woman is greater
# 95 percent confidence interval:
#  -109.5565       Inf
# sample estimates:
#   mean in group man mean in group woman
#                46.5                36.0
```

How would you interpret this result?

- (A) There are no significant gender differences in technological skill.
- (B) Gender influences the thechnological skill of a person.
- (C) Women have significantly more tech skills than men.

- (D) T-Tests are confusing.

8.4 ANOVA

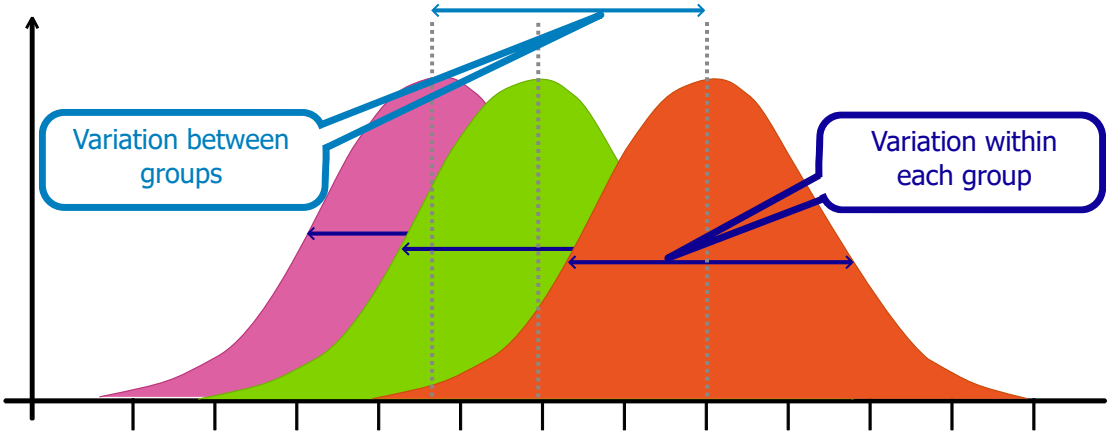


Figure 8.3: ANOVA principle

- Like a t-test for more than two groups
- Why do we not just calculate several t-tests?
  - α inflation!
  - Significance level of 0.05 means that 1/20 tests will be significant by pure chance, so more tests makes it more likely that we hit that chance and make an alpha error (falsely reject null hypothesis)

8.4.1 How to - theoretically

1. Check Assumptions
   - Data should be normally distribution & variance in groups should be similar (homogeneous)
2. Sum of Squares: Sum of Squares total, within & between (*R does this for us*)
   - F-fraction as the measure of variance explained by the grouping variable in comparison to other variability in the dependent variable

3. Interpretation and post-hoc tests
    - If there are any significant differences at all, we can use pairwise t-tests (with alpha correction!)

Example: Music genre and loudness

1. Check assumptions

- Check for Homogeneity of Variance with the Levene Test
- ```
  # Make sure the package "car" is installed first! If not, install.packages("car")
  # as.factor() forces R to recognize our group as such!
  car::leveneTest(v08_loudness ~ as.factor(v07_genre), data = seminar, center = mean)
  # Levene's Test for Homogeneity of Variance (center = mean)
  #       Df F value Pr(>F)
  # group  2  0.1729 0.8437
  #       10
  ```
- Interpretation?
    - p value < 0.05 would indicate significant differences in variance between the group, so we want it to be > 0.05
    - Assumption met!

2. Define the overall model

```
model <- aov(v08_loudness ~ as.factor(v07_genre), data = seminar)
summary(model) # "Pr(>F)" is the p-value
#                      Df Sum Sq Mean Sq F value Pr(>F)
# as.factor(v07_genre)  2  618.9   309.4   2.562  0.126
# Residuals            10 1208.0   120.8
```

- Interpretation?
    - Not significant (likely due to small sample size)
    - usually we would stop here then, but we will look at the post hoc tests anyway ;)
```

3. Post Hoc Test

```
TukeyHSD(model)
#   Tukey multiple comparisons of means
#     95% family-wise confidence level
#
# Fit: aov(formula = v08_loudness ~ as.factor(v07_genre), data = seminar)
#
# $`as.factor(v07_genre)`
#                   diff        lwr      upr     p adj
# rock-pop     12.566667  -5.677791 30.81112 0.1922467
# techno-pop   17.400000  -7.808341 42.60834 0.1911143
# techno-rock   4.833333 -19.767489 29.43416 0.8544384
```

- Interpretation?
  - There are no significant pairwise differences in our (small) sample.
  - But we can simulate a larger sample (for fun)

8.4.2 Addendum for demonstration only: Bootstrapped Data for larger sample size

```
data <- data.frame()

for(i in 1:10){
  boot <- seminar[sample(1:nrow(seminar), nrow(seminar), replace = T), ]
  data <- rbind(data, boot) # create many random samples from our data
}

bootstrapped_model <- aov(v08_loudness ~ as.factor(v07_genre), data = data)
summary(bootstrapped_model)
#                       Df Sum Sq Mean Sq F value   Pr(>F)
# as.factor(v07_genre)   2   7345    3672   47.76 3.42e-16 ***
```

```
# Residuals             127   9766      77
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
TukeyHSD(bootstrapped_model)
#   Tukey multiple comparisons of means
#     95% family-wise confidence level
#
# Fit: aov(formula = v08_loudness ~ as.factor(v07_genre), data = data)
#
# $`as.factor(v07_genre)`
#                   diff       lwr       upr     p adj
# rock-pop    14.238278 10.307525 18.169030 0.0000000
# techno-pop  17.624242 11.977093 23.271392 0.0000000
# techno-rock  3.385965 -2.236703  9.008633 0.3295653
```

8.5 Reporting with the apa & papaja packages

- You know the papaja package already for theme_apa() in data visualization
- The package also has many wrapper functions to make reporting in R & R Markdown a lot easier
  - "apa_print()"
  - Chi^2 Test reporting cannot be achieved with this, so we use apa::chisq_apa() for that

8.5.1 Usage

```
# Chi (add format = "rmarkdown" if needed)
apa::chisq_apa(chisq.test(seminar$v07_genre, seminar$soul_dummy))
# Warning in chisq.test(seminar$v07_genre, seminar$soul_dummy): Chi-squared
# approximation may be incorrect
# chi^2(2) = 2.14, p = .344
```

```
# t-test
papaja::apa_print(t.test(v05_skill_tech ~ v01_gender, data = seminar, alternative = "greater"))$full_re
# [1] "$\\Delta M = 10.50$, 95\\% CI $[-109.56, \\infty]$, $t(1.28) = 0.39$, $p = .377$"

# ANOVA
papaja::apa_print(model)$full_result |> suppressMessages()
# $as_factorv07_genre
# [1] "$F(2, 10) = 2.56$, $\\mathit{MSE} = 120.80$, $p = .126$, $\\hat{\\eta}^2_G = .339$"
# suppressMessages() is a useful helper function that avoids any messages from the function it is used
```

8.5.2 Usage in R Markdown

- This presentation is based on R Markdown, so we can make use of the pretty printing options right here
- By using `apa_print(model)$full_result`, we can automatically report results inside our documents:
- "In our sample, ANOVA showed no significant differences between preferred music genre and preferred volume of listening to music ($F(2, 10) = 2.56$, $MSE = 120.80$, $p = .126$, $\hat{\eta}^2_G = .339$). However, bootstrapping with 10 repetitions suggests that this lack of evidence might be due to the small sample size ($F(2, 127) = 47.76$, $MSE = 76.90$, $p < .001$), which is also supported by the large effect size ($\hat{\eta}^2_G = .429$)."

Wrap-Up & Further Resources

Chi^2 test measures association between two categorical variables

t Test measures differences between mean values (one sample, two sample, paired)

ANOVA can be thought of as an augmentation of the t test while controlling alpha inflation

Functions: chisq.test(), t.test(), aov()

Always try to imagine/ keep in mind what you might expect and *what the data would be like if that*

were true

Read the documentation of each function for more options

Statistics Picker

Chi2-test (Statology)

t-test (Statology)

ANOVA (Statology)

Discovering Statistics Using R (**?**)

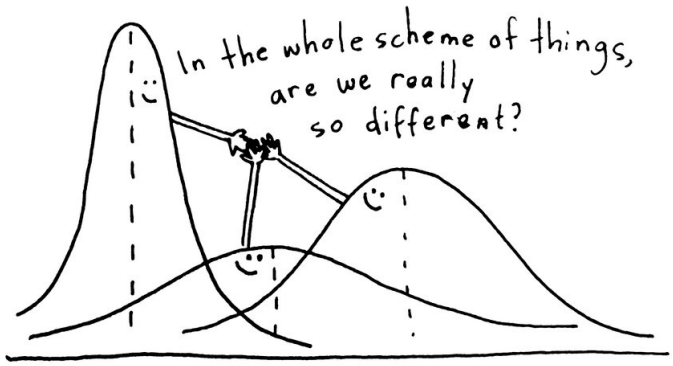


Figure 8.4: Cute ANOVA curves

https://www.pinterest.de/pin/59180182590147005/

Chapter 9

Association Statistics

In this chapter we will look into the correlation and linear regression model in R.

I will assume that you have a basic understanding of what all statistics mentioned in this book are, as they are commonly used among psychologists. However, let's start with a brief statistics re-cap to be on the same page.

Both the correlation - usually reported with Pearson's *r* coefficient - and the regression - also known as (general) linear model - give us measures of **association**. The association can be positive (the larger x, the larger y) or negative (the larger x, the smaller y), which is reflected by the sign of the coefficient: *r = 1* would be a perfect positive correlation and *r = -1* would be a perfect negative correlation.

Hopefully, you have heard this one thousand times already, so let me tell you for the thousand and first time: **Correlation does not imply causation**. The correlation coefficient r only tells us about the direction and strength of the association and nothing about the causal relation. However, in the regression model we can check whether Y changes on the basis of X, so we can include assumptions of cause and effect in our model[1]

[1] A linear model that is ill-defined might still become significant and thus misleading, so your modeling decisions should always be made with a theoretical basis.

Figure 9.1: Age and circumference of orange trees shown in a scatterplot. We can see that there seems to be a positive relationship between the two measures: The older the tree, the larger the circumference.

9.1 Correlation

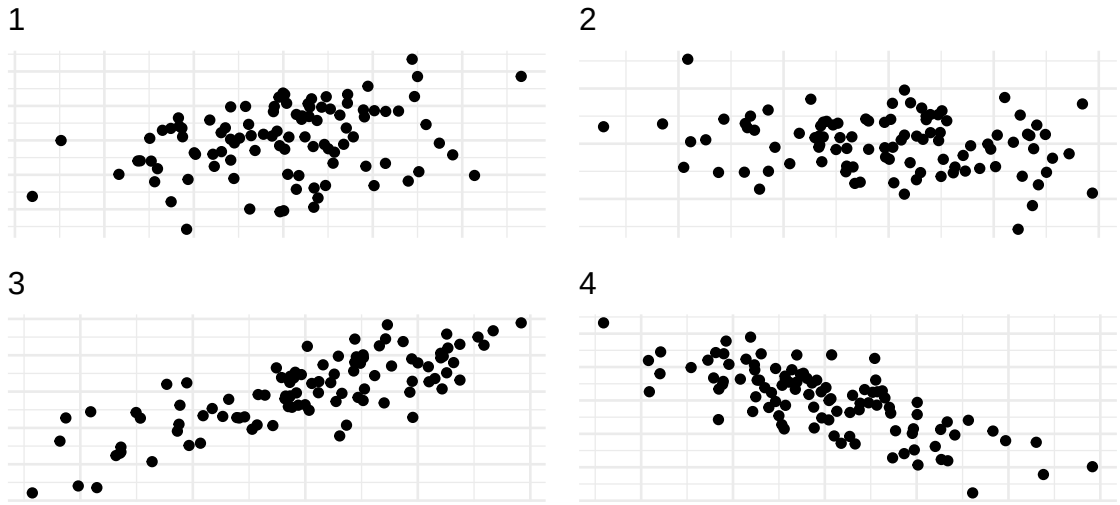
The pearson correlation coefficient r measures **association between two numeric variables**. The variables need to

- be *continuous* & interval-scaled
- be *normally distributed* & should have *no outliers*
- have a *linear relationship*.

Its range is from -1 to 1 which would mean a perfect negative or positive association, respectively. The closer r is to 0, the weaker the correlation.

Since there will usually be some variation and noise in the data, I believe it makes sense to get a feeling for what different associations might look like. So, you can play a game of "guess the correlation": Look at the four plots below and guess for each how strong the correlation could be be! Remember - a **positive** correlation means "**the more X, the more Y**" and a **negative** correlation means "**the more X, the less Y**".

9.1.1 Guess the correlation!



Solution

r1 = 0.3, r2 = -0.2, r3 = 0.8, r4 = -0.75

9.2 Correlation in R

- Two main functions:
  - `cor()` calculates the correlation
  - `cor.test()` calculates correlation and significance
- As input they both need only an x and a y variable
  - You can specify some other aspects of the calculation, such as statistical method (e.g. "spearman") or how to deal with missing data

```
x <- 1:10
y <- sample(x, 10)
cor(x, y)
# [1] -0.3333333
cor.test(x, y)
#
#   Pearson's product-moment correlation
```

```
#
# data:  x and y
# t = -1, df = 8, p-value = 0.3466
# alternative hypothesis: true correlation is not equal to 0
# 95 percent confidence interval:
#  -0.7959163  0.3749953
# sample estimates:
#       cor
# -0.3333333
```

9.2.1 Handling missing data

- Many functions have an option for missing data or `NA`s
  - You can often add the argument `na.rm = TRUE` to a function for "NA remove"
- In the `cor()` function, we define to only use complete observations
- ```
  k <- c(1, 2, 3, 4, 5)
  m <- c(1, 3, 2, 5, NA) # same length but 1 data point is missing
  cor(k, m)
  # [1] NA
  ```
- With `use = "complete.obs"` we define to only use pairs of observations that are not missing
- ```
  cor(k, m, use = "complete.obs")
  # [1] 0.8315218
  ```
- ```
  cor(k[1:4], m[1:4])
  # [1] 0.8315218
  ```

9.2.2 Exercise

Is technology skill associated with seminar motivation?
  ```

Calculate a correlation test using `cor.test()` to analyze the question.

Try to formulate an interpretation as you would report it in a thesis or paper!

Solution

```
cor.test(seminar$v05_skill_tech, seminar$v10_motivation)
#
#   Pearson's product-moment correlation
#
# data:  seminar$v05_skill_tech and seminar$v10_motivation
# t = 1.386, df = 11, p-value = 0.1932
# alternative hypothesis: true correlation is not equal to 0
# 95 percent confidence interval:
#  -0.2100186  0.7724602
# sample estimates:
#       cor
# 0.3855856
```

"With r = 0.386 there is a positive association of moderate strength between previous technological skill and motivation for the seminaR. This association is not significant (p = 0.193), likely due to the small sample size."

9.2.3 Quiz

Look at our seminar dataset by entering `str(seminar)` in the console. Which of these correlations would work? Choose "TRUE" if you think the correlation would work and "FALSE" if you think it would not. You can look at the explanation for those that would not work below!

1. `cor(seminar$v02_age, seminar$v04_bodyheight` TRUE / FALSE
2. `cor(seminar$v02_age, seminar$v08_loudness)` TRUE / FALSE
3. `cor(seminar$v08_loudness, seminar$v06_loc)`TRUE / FALSE

Explanation

1. There is a closing bracket missing

2. The variable *v06_loc* has numbers, but they are recognized as characters!

9.3 Linear Regression

- Linear regression also works on numerical, normally distributed data
- We assume an association, and regression can help to look for causation
  - There is one **dependent variable** y and one **independent variable** x
  - In multiple linear regression, there can be several x
- Formula:

$$y = \beta_0 + \beta x + \epsilon$$

- What we are essentially doing is building a model for our data and checking how well it actually fits!

9.3.1 Build the model

- The R function for regression analysis is `lm()` for *linear model*
- It needs a "formula" as input - similar to the formula in the `t.test()`, we need the ~
  - Read Y ~ X as "Y on the basis of/ given X"
  - Our dependent variable Y goes first and our independent variable(s) go after the ~
- If the variables come from a data set, we need to specify data as well

9.3.2 Visual Inspection

```
ggplot(Orange, aes(x = age, y = circumference, color = age)) + geom_jitter(size = 3) +
  geom_smooth(method = "lm", se=FALSE, color="lightgray",
              linewidth = .7, formula = y ~ x) +
  theme_minimal() +
  labs(x = "Tree Age", y = "Trunk Circumference", title = "Do trees get thicker with age?") +
  scale_color_distiller(palette = 7) +
  theme(legend.position = "none")
```

What could be problematic here?

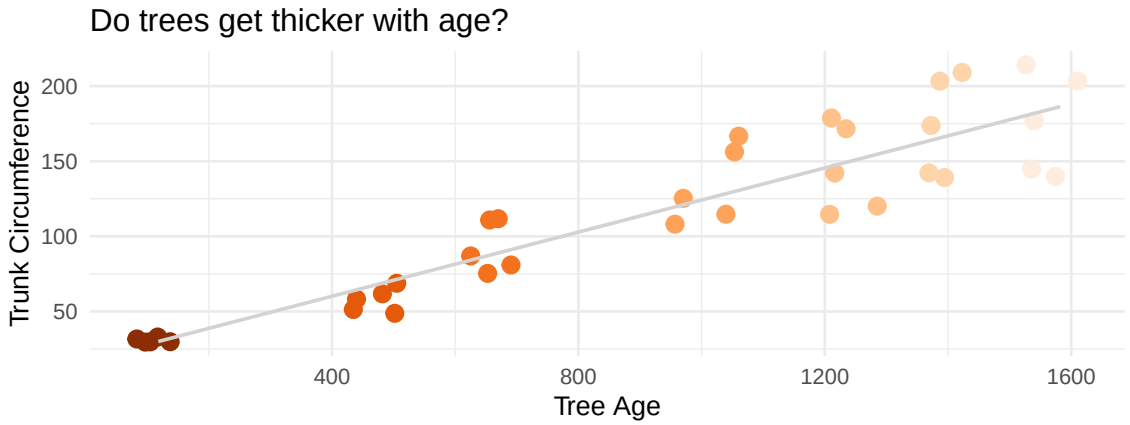


Figure 9.2: Scatterplot of orange tree age and cirumference from before with the regression line added. Notice that the variable which is used as the independent X (age) is plotted on the X axis for intuitive reading of the plot.

- (A) homoscedasticity
- (B) multicollinearity
- (C) heteroscadasticity
- (D) oranges

9.3.3 Build the model

9.3.3.1 Do trees get thicker with age?

```
Treelm <- lm(formula = circumference ~ age, data = Orange)
Treelm
#
# Call:
# lm(formula = circumference ~ age, data = Orange)
#
# Coefficients:
```

```
# (Intercept)          age
#     17.3997       0.1068
```

- The lm alone gives us the mathematical formula
- To look at the statistical results, we need to use another function such as `print()` or `summary()`

9.3.4 Analyze the model

```
summary(Treelm)
#
# Call:
# lm(formula = circumference ~ age, data = Orange)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -46.310 -14.946  -0.076  19.697  45.111
#
# Coefficients:
#              Estimate Std. Error t value Pr(>|t|)
# (Intercept) 17.399650   8.622660   2.018   0.0518 .
# age          0.106770   0.008277  12.900 1.93e-14 ***
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 23.74 on 33 degrees of freedom
# Multiple R-squared:  0.8345,  Adjusted R-squared:  0.8295
# F-statistic: 166.4 on 1 and 33 DF,  p-value: 1.931e-14
```

- Interpretation?
  - Trees get larger circumferences the older they are, but this might be modulated by their Species, environment or other factors

9.3.5 Exercise

Does the age of a person have an influence on how long they took to complete the seminar survey (session length)?

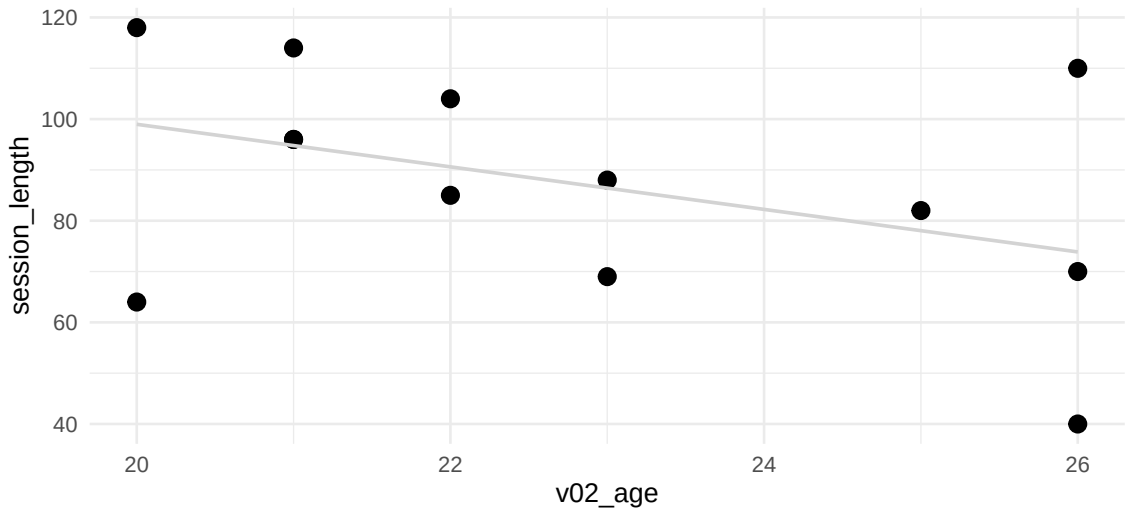
Use the `lm()` function and report the significance level of the predictor as well as the model equation.

Solution (code)

```
age_sess <- lm(session_length ~ v02_age, data = seminar)
summary(age_sess)
#
# Call:
# lm(formula = session_length ~ v02_age, data = seminar)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -34.978  -5.605   1.209  13.395  36.141
#
# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
# (Intercept)  182.706     61.517   2.970   0.0127 *
# v02_age       -4.186      2.689  -1.557   0.1478
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#
# Residual standard error: 21.23 on 11 degrees of freedom
# Multiple R-squared:  0.1805,  Adjusted R-squared:  0.106
# F-statistic: 2.423 on 1 and 11 DF,  p-value: 0.1478
```

Solution (interpretation)

"The age of a person does not significantly predict the time it took them to complete the survey (p = .148). The model equation is 182.7 -4.19X with age explaining about 18% of the variance in session length for the survey."



9.3.6 A word to the wise

- There is also a function called `glm()` for general linear model
- In the cases I showed you, both perform the same tasks
- The `glm()` can also handle other more advanced statistical analyses, including logistic regression
- However, the `lm()` function will output the coefficient of determination R^2
  - It tell us the proportion of the variation in the dependent variable that is predictable from the independent variable(s)
  - We *could* also calculate it by hand using the `cor()` function and squaring the result

9.4 ANOVA Exercise

Reminder: There are generally 3 steps to an ANOVA

```
car::leveneTest(v08_loudness ~ v11_soul, data = seminar) # 1.
model <- aov(v08_loudness ~ v11_soul, data = seminar) # 2.
summary(model)
TukeyHSD(model) # 3.
```

1. Check assumptions with Levene Test
2. Build the model to perform an omnibus ANOVA
3. Perform post-hoc tests to check pairwise differences (usually only if the omnibus ANOVA is significant)

9.4.1 Exercise

Does the preferred music volume depend on someone's soul philosophy?

Perform an ANOVA on our seminar data to explore the question (v08 & v12).

Solution omnibus ANOVA

```
car::leveneTest(v08_loudness ~ v12_soul_phil, data = seminar)
# Warning in leveneTest.default(y = y, group = group, ...): group coerced to
# factor.
# Levene's Test for Homogeneity of Variance (center = median)
#       Df F value Pr(>F)
# group  2  0.3112 0.7394
#       10
model <- aov(v08_loudness ~ v12_soul_phil, data = seminar)
summary(model)
#               Df Sum Sq Mean Sq F value  Pr(>F)
# v12_soul_phil  2 1164.8   582.4   8.797 0.00625 **
# Residuals     10  662.1    66.2
# ---
# Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Look for pairwise differences even if the overall ANOVA does not reach significance.

Solution pairwise comparison

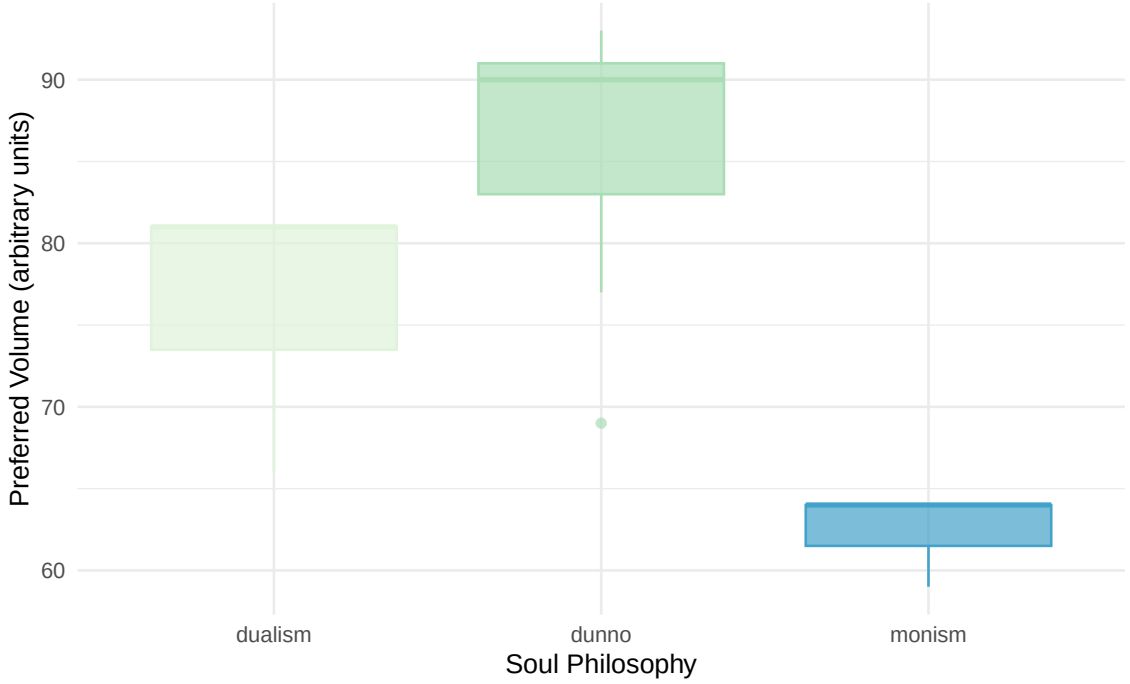
```
TukeyHSD(model)
#   Tukey multiple comparisons of means
#     95% family-wise confidence level
```

```
#
# Fit: aov(formula = v08_loudness ~ v12_soul_phil, data = seminar)
#
# $v12_soul_phil
#                       diff        lwr       upr     p adj
# dunno-dualism     9.714286  -5.678116 25.106687 0.2419042
# monism-dualism -13.666667 -31.879202  4.545868 0.1490289
# monism-dunno   -23.380952 -38.773354 -7.988551 0.0049965
```

Choose and create an appropriate visualization for this data!

Solution Data Viz

```
ggplot(seminar, aes(x = v12_soul_phil, y = v08_loudness,
                    color = v12_soul_phil, fill = v12_soul_phil)) +
  geom_boxplot(alpha = .7) +
  theme_minimal() +
  theme(legend.position = "none") +
  labs(x = "Soul Philosophy", y = "Preferred Volume (arbitrary units)") +
  scale_color_brewer(palette = 4) + scale_fill_brewer(palette = 4)
```



Wrap-Up & Further Resources

Correlation coefficient r can be determined using `cor(x,y)`

R^2 is the coefficient of determination in a linear model (calculate by hand or in the model formula)

The linear model function `lm()` is used to build models for linear regression

Problems such as overfitting or heteroscedasticity reduce the interpretability of the model results

Guess the Correlation

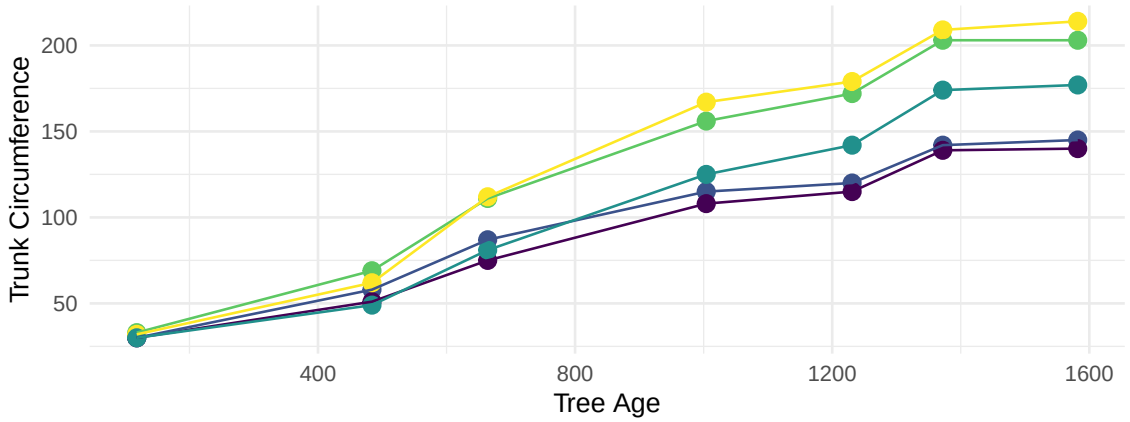
Explanation: Correlation

Linear Regression in R

lm() cheatsheet

```
ggplot(Orange, aes(x = age, y = circumference, color = Tree)) +
  geom_point(size = 3) + labs(x = "Tree Age", y = "Trunk Circumference") +
```

```
  geom_line(aes(color = Tree)) + theme_minimal() + theme(legend.position = "none")
```



Part III

Reporting in R