# Data Visualization for Communication Science

Morley J Weston

2025-02-04

# Table of contents

# Preface

This is the class textbook for the course "Data Analysis and Visualization for Communication Science" at the University of Zurich.

As I am adapting this course to work online, I will be updating the book as I go along, so expect changes and updatesIt is written in Markdown and compiled using the Quarto toolchainThe source code is available on GitHub at https://github.com/morleyjamesweston/data_viz_class_spring_2025).

## Contents

### Module 1: Working with data

#### Week 1: Tidyverse 1: Data wrangling *(19-02-2025)*

- Basic R programming
- Filtering & Summarizing data

#### Week 2: Tidyverse 2: Data transformation *(26-02-2025)*

- Importing data
- Shaping data
- Joining data

#### Week 3: Tidyverse 3: Data tips & Tricks *(05-03-2025)*

- Data cleaning practice
- Dealing with missing data
- Dates & times

#### Week 4: Finding your own datasets *(12-03-2025)*

- Where to find data sources
- Data cleaning

## Module 2: Data visualization

### Week 5: GGplot 1: Basic charts and graphs *(19-03-2025)*

- Basic plots
- Customizing plots
- Saving plots

### Week 6: GGplot 2: Making it look good *(26-03-2025)*

- Themes & colors
- Fonts & labels
- Faceting

### Week 7: GGplot 3: advanced charts and graphs *(02-04-2025)*

- Beyond bars, lines, and points
- Using the grammar of graphics

### Week 8: Aesthetics *(09-04-2025)*

- Color theory
- Accessibility
- Web colors
- Publishing your work

### Midterm Presentations *(16-04-2025)*

### Easter break: No class *(23-04-2025)*

## Module 3: Advanced topics

### Week 9: Maps & geospatial data *(30-04-2025)*

- Using geospatial data
- Creating maps

### Week 10: Interactivity & the internet *(07-05-2025)*

- Creating interactive visualizations with Plotly
- Publishing notebooks and websites

### Week 11: Learning on your own *(14-05-2025)*

- Finding libraries
- Reading documentation
- Beyond R and GGplot
- Next steps

### Week 12: Tables & statistics *(21-05-2025)*

- Exporting and submitting statistical tables
- Plotting statistical results
- Visualizations for academia

### Final presentations *(28-05-2025)*

---

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

# Before we start: Installing software

For this course, we'll primarily be using R and RStudio to create our data visualizations. R is a programming language, and RStudio is an application that makes it easier to write and run R code.

You can access R and RStudio in two ways: using the class server, or installing it on your own computer. Here are some instructions for both:

## The easy way: Using the class server

The easiest way to get started is to use the class server. Signing up will allow you to access Rstudio via your web browser, and has everything set up for you. You can find the link to the class server in the course syllabus, on OLAT, and I've also sent you an email.

If you're just starting out, or if you have an older computer, I'd recommend taking this route.

Once you've signed up, you have one more 5-minute assignment, detailed at **?@sec-keyboard-homework**.

## The hard way: Installing on your own computer

Alternatively, you can install R and RStudio on your own computer. This will let you keep all the data, code, and packages you install, and you can continue using R after the class is over. Here's how you can do it:

### Installing R

1. Download R from https://cran.rstudio.com/.
2. Follow the instructions for your operating system.

## Installing RStudio

1. Download RStudio from: https://posit.co/download/rstudio-desktop/
2. Follow the instructions for your operating system.
3. Now open up RStudio, and you should see something like this:

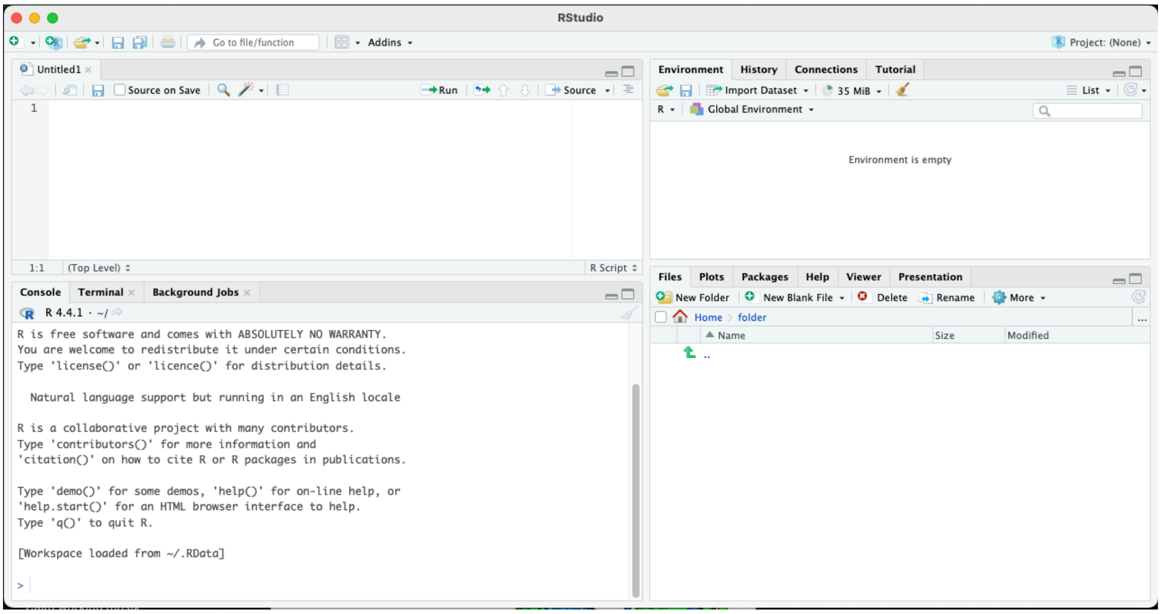



Figure 1: Congratulations! You've installed RStudio.

## Testing your installation

Let's make sure you have the right version of R installed. Find the window called "Console" in RStudio. By default, it is in the bottom left of the screen. Type the following into the console, and hit "Enter".

```
R.version.string
```

If you've done everything correctly, it will be 4.3 or 4.4.

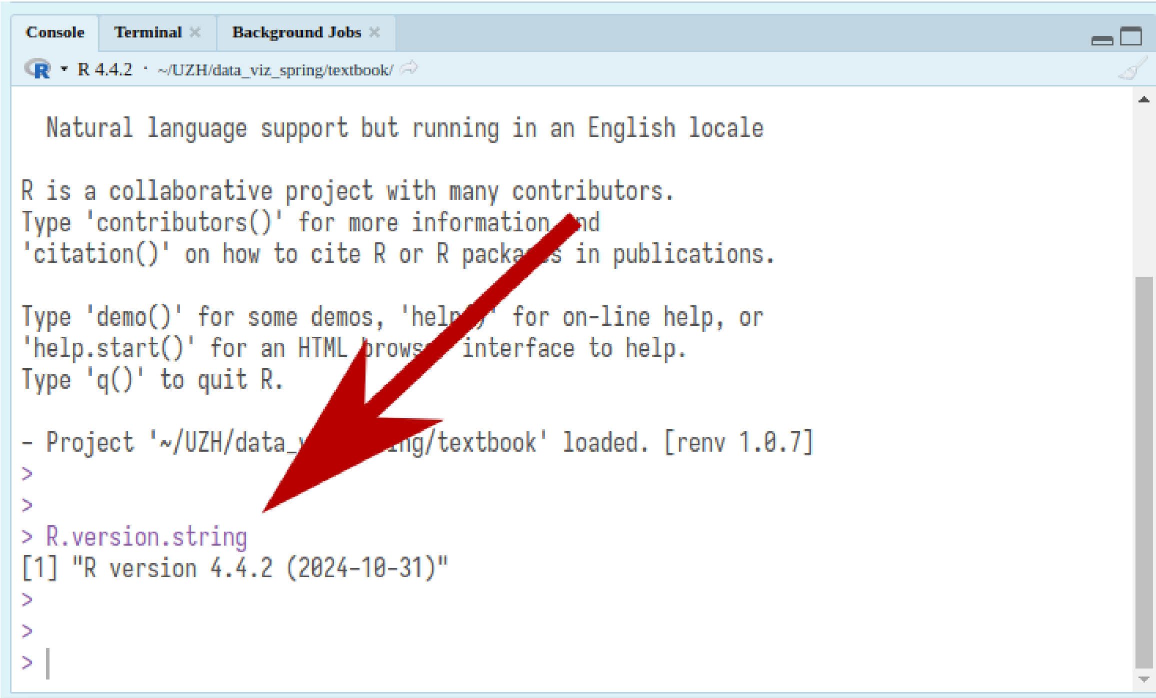



Figure 2: The output should look something like this.

### Setting up RStudio

Before we start, let's fiddle around with some settings. In the menu at the top of the screen, go to the "Tools" > "Global Options" menu. Then open up the "Code" tab.

Check the box that says "Use native pipe operator".

Don't worry about what this means for now. You'll know all about the pipe operator by the end of this course.

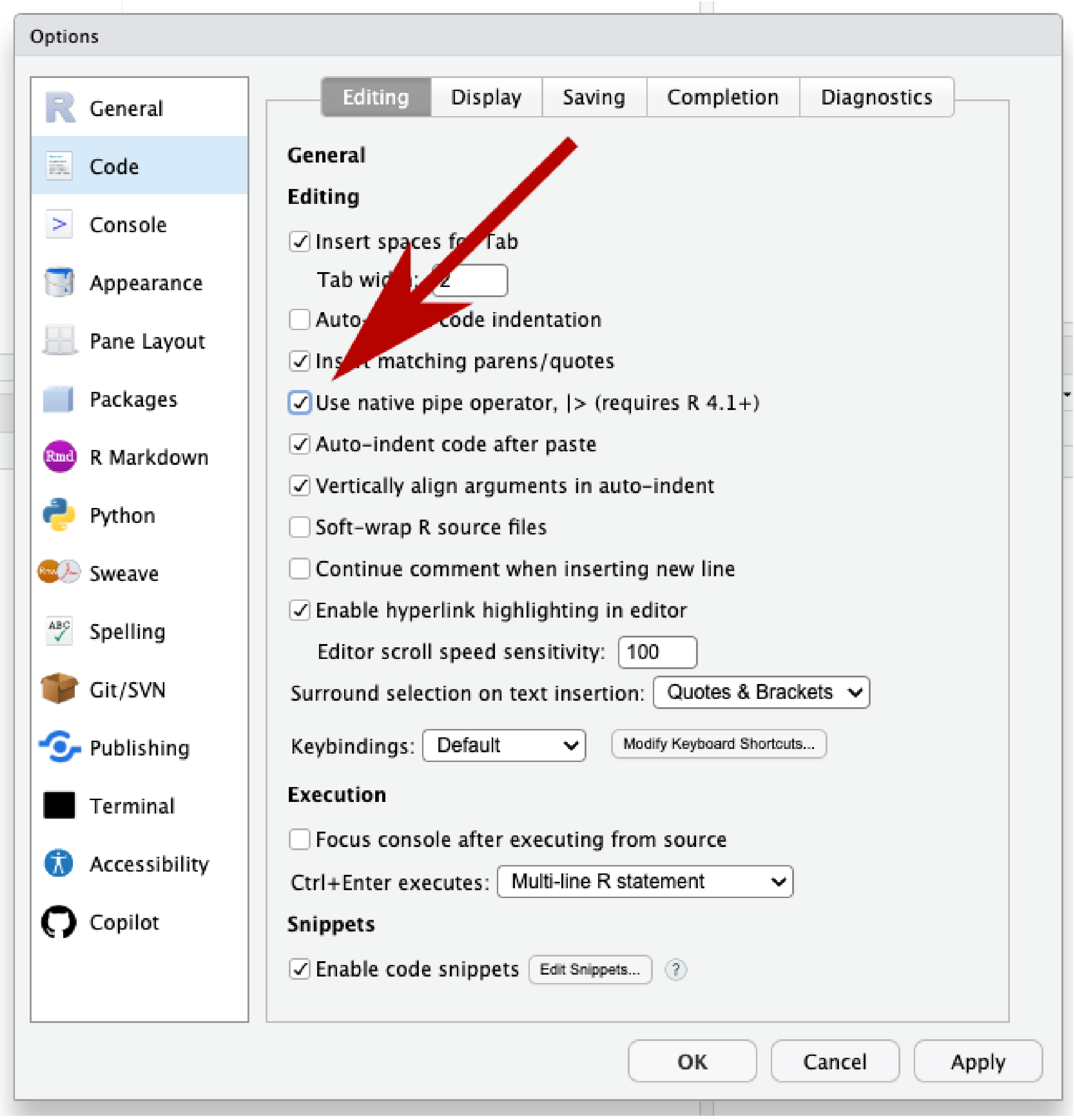



Figure 3: The option is in the "Code" tab.

While not directly covered in this course, also note the "Git" and "Copilot" tabs if you're already familiar with them. These are very useful tools for working with code, and you should definitely check them out.

## Installing the Tidyverse

For this class, we'll be relying heavily on a package called the Tidyverse, short for "Tidy Universe". It is, in my opinion, the one thing that makes R better than any other language for data analysis and visualization.

To install the Tidyverse, type the following into the console:

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

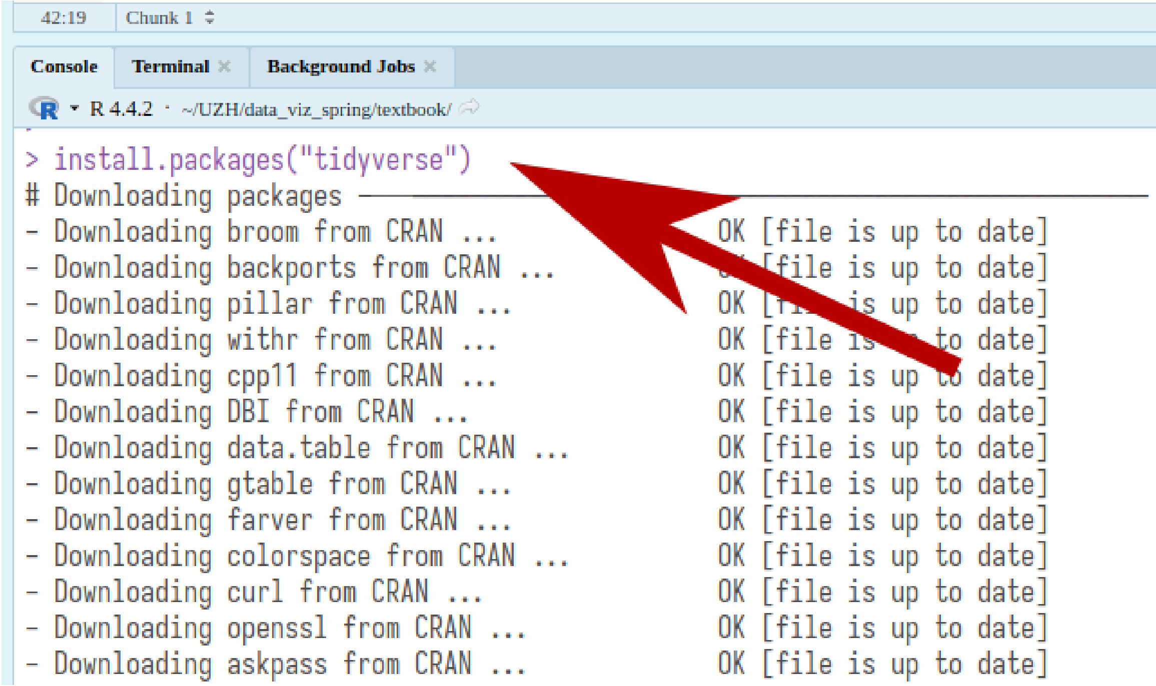



Figure 4: install.packages("tidyverse") will download the Tidyverse package. This is kind of like installing a new app onto your computer.

You'll be prompted with an option to type Y or N.

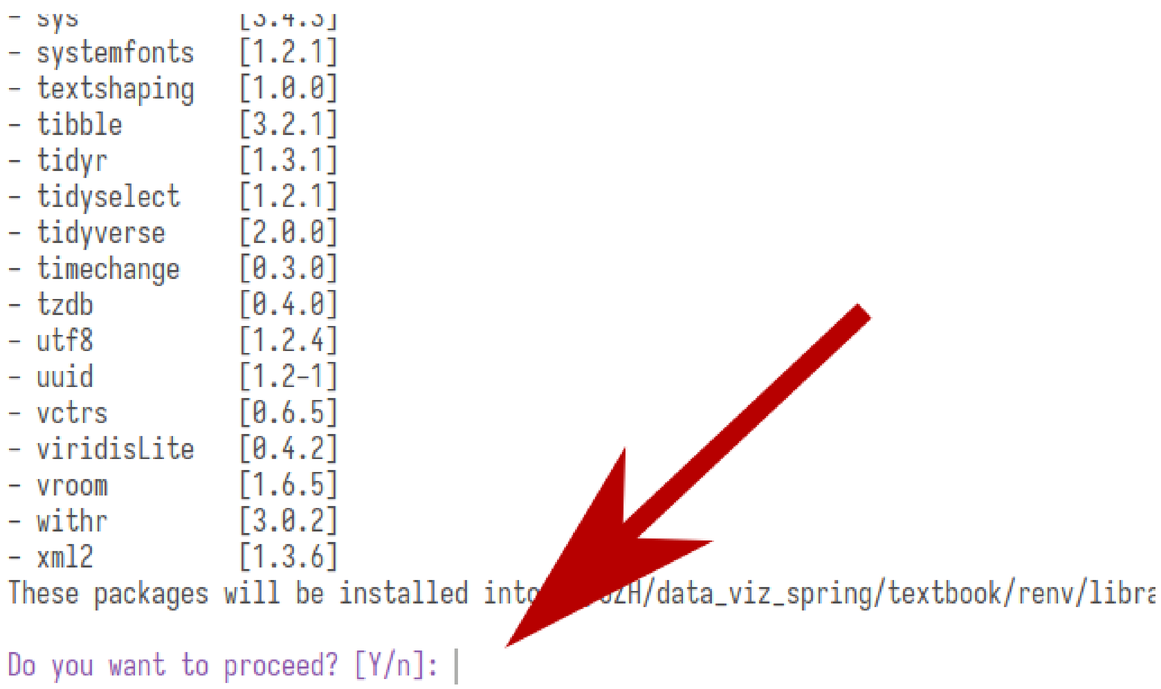



Figure 5: Type Y for yes, you want to install.

To make sure it installed correctly, type the following into the console:

```
library(tidyverse)
```

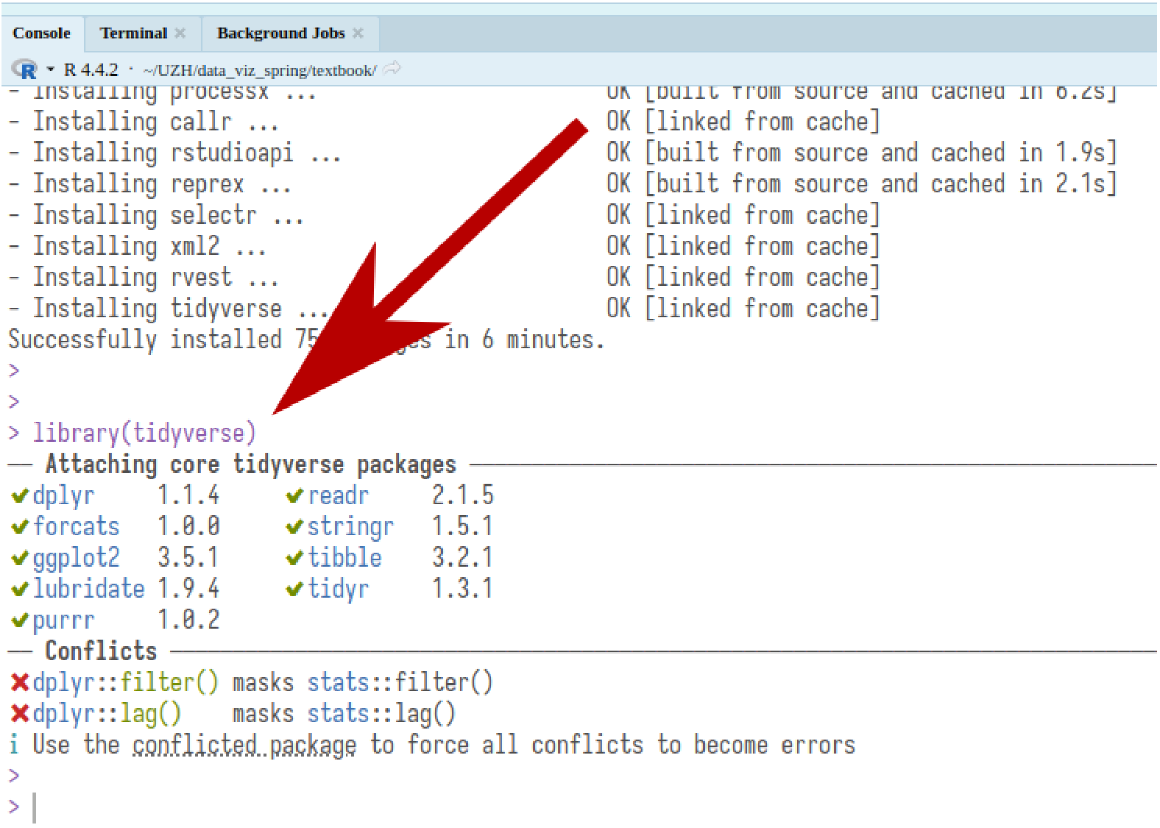



Figure 6: library(tidyverse) will load the package; this is kind of like opening an app on your computer.

### Homework: Getting to know your keyboard

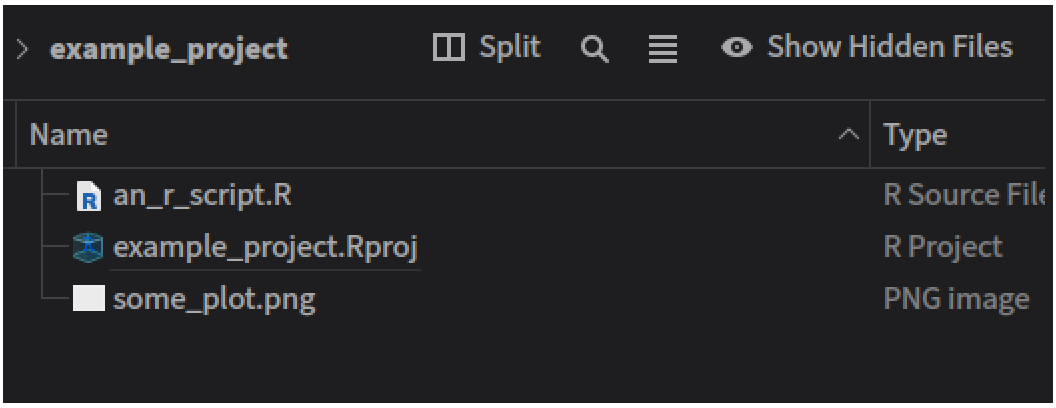
This is a very international class, and we all have slightly different keyboards. Your first assignment is to figure out how to type the following keys:

```
& $ | [] {} () \ / ~ ` ^ < > %
```

# 1 Tidyverse 1: Data wrangling

## 1.1 Setting up a Project

The first thing you should do when you start a new project is to create a new project in RStudio. This will keep all of your files and data in one place, and easy to find later.

> **i** What is a project?
>
> A project is a way to organize your work in RStudio. It keeps all of your files and data in one place, and makes it easy to find later. You should create a new project for each new project you start. For this class, feel free to use the same project for all of the lessons.
> This will also create a new folder on your computer where all of your stuff lives, and you should know how to find it outside of RStudio. If you're using the class server, the folder will be on a university computer, and you won't have access to it outside of RStudio.
>
> 
> 
>
> Figure 1.1: A project is just a folder with a special file (.Rproj) inside it. It is where all the stuff you make will be.

To create a new project, go to "Project" in the upper right hand pane and select > "New Project".

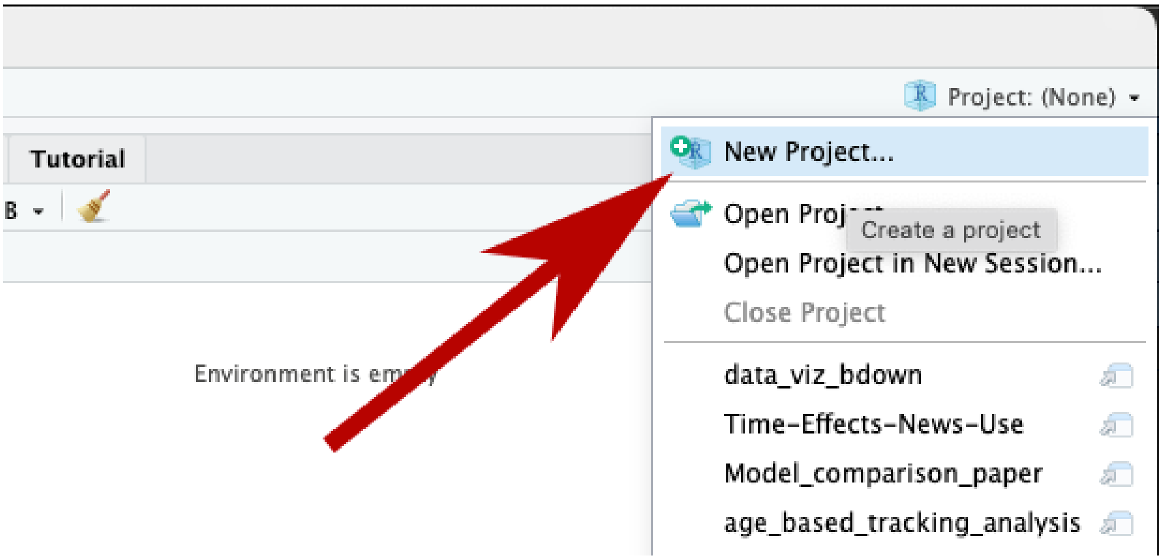



Figure 1.2: You can also use this dropdown menu to switch between projects.

You should save it in a new directory (another word for folder).

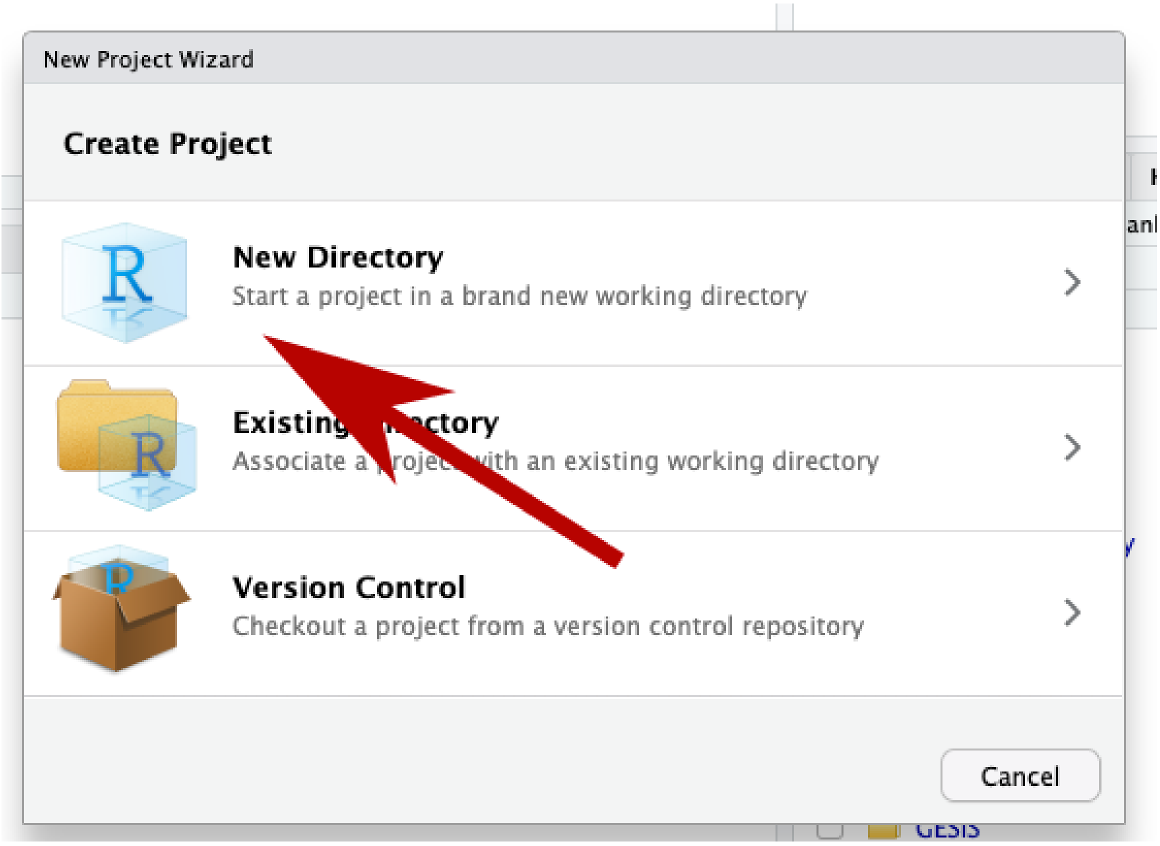



Figure 1.3: Select the “New Directory” option

You'll be presented with a list of options, but for this class I'd go with "New Project".

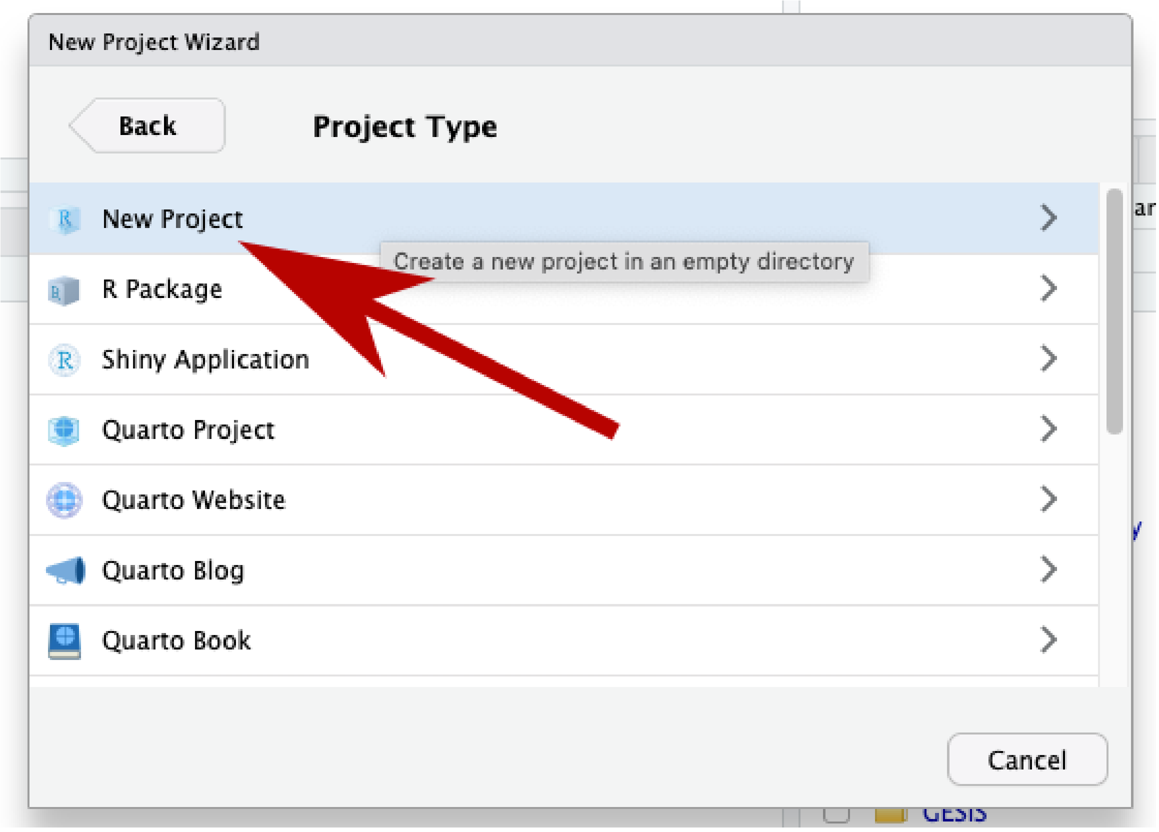



Figure 1.4: You can use R to do a lot of different things, such as build a website or write this workbook.

Finally, You can call it whatever you like and save it wherever you like, but I suggest something like "data-viz-class", and maybe put it in your "Documents" folder.

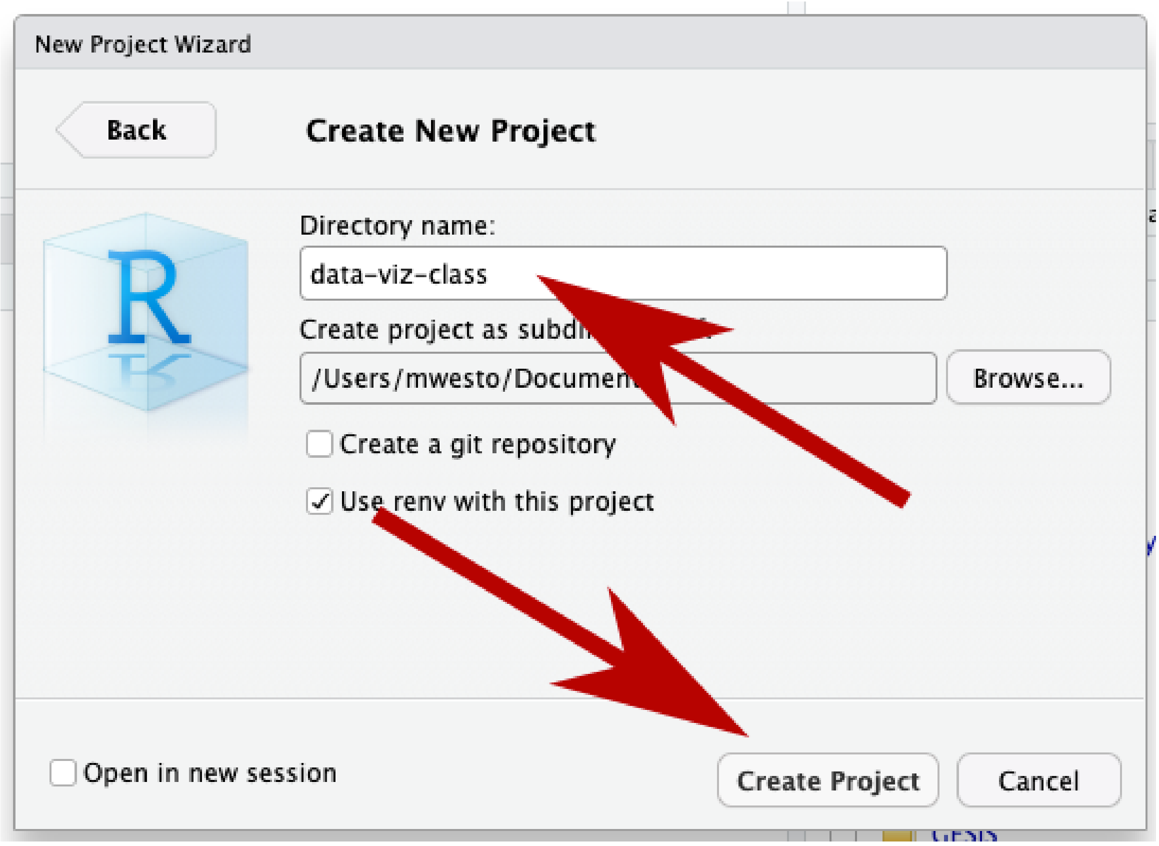



Figure 1.5: You'll be using this project for the whole class, so give it a name you'll remember.

Now hit "Create Project", and we can start!

## 1.2 Making a new file

Every time we write a program in R, we should put it in its own file. A good first step for today's work is to make a new R script via:

File > New File > R Script

Give it a name you'll remember later like *week_1_intro.R*

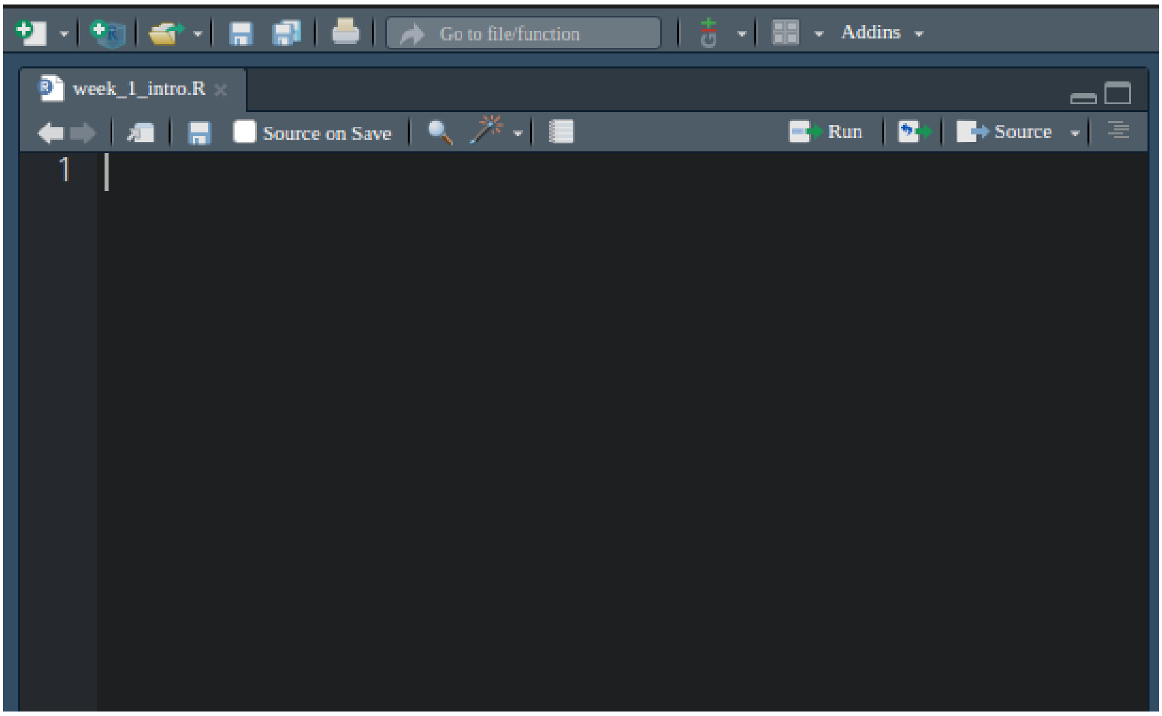



Figure 1.6: You now should have a new file that looks like this.

This is … just an empty text file. Very underwhelming. But this is where we're going to write our code, and make some interesting things happen.

## 1.3 Loading the Tidyverse package

For this class, we're going to use a package called the Tidyverse. This is a collection of packages that make it easier to work with data. For this, and all of our classes, you'll want to add this line to the beginning of your script:

```
library(tidyverse)
```

> **What is 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." R has always been one of the best tools for doing statistics, but handling the actual data was always kind of a mess, and this largely fixes it. Simply put, it makes R much better at doing data science, and is easily my favorite tool in this space.

## 1.4 Running code

Now, we've written our first line of code. But how to run it? You have two options.

- First is to use the "Run" button at the top of the script. This will run the line of code that your cursor is on.

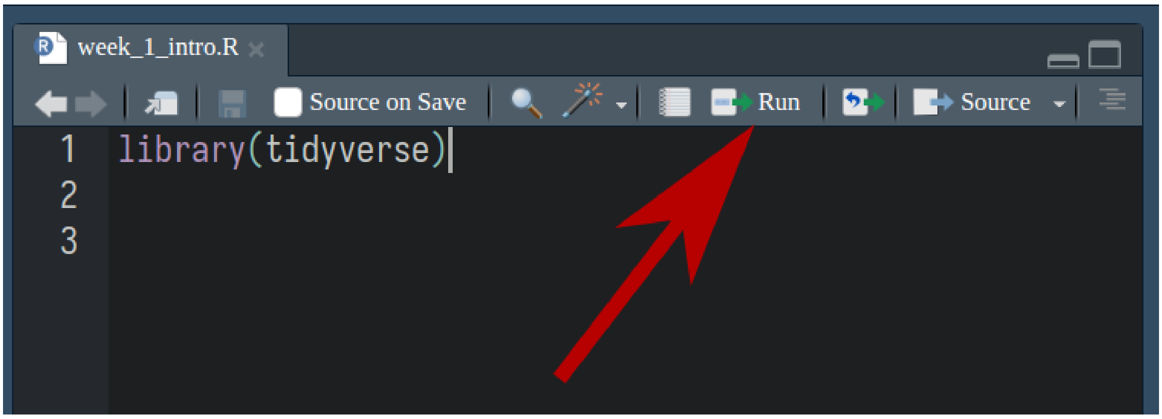



Figure 1.7: Hit that run button!

- Second is to use the keyboard shortcut Ctrl-Enter. This will also run the line of code that your cursor is on, and is the option I typically use.

When you do this, you'll (hopefully) see a bunch of messages like this in the console down below:

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.1     v tibble    3.2.1
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.0.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to be
```

These messages are nothing to worry about, they're just telling you what new tools you have because you loaded the Tidyverse package.

## 1.5 Downloading data

For this lesson, we're going to look at the names of horses in Switzerland, A very important topic that affects all of our lives. The data set can be found at:

https://tierstatistik.identitas.ch/en/equids-topNamesFemale.html

https://tierstatistik.identitas.ch/en/equids-topNamesMale.html

### 1.5.1 Making a new folder on your computer in RStudio

Before we download the data, we should make a new folder to put it in. This will keep our project organized, and make it easier to find things later.

In the lower right hand corner, you'll see a "Files" tab. This has a button that looks like a folder with a plus sign on it. Click this to make a new folder, and call it `input_data`.

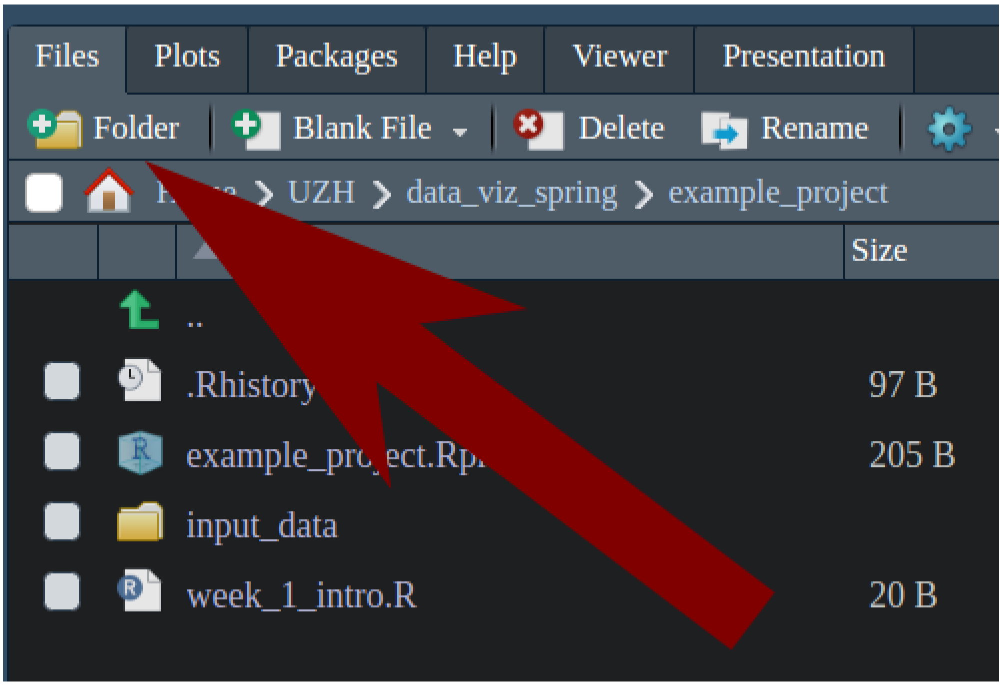



Figure 1.8: Make a new folder using this button

### 1.5.2 Downloading files using `download.file()`

The first thing we need to do is download the data. We can do this with the `download.file()` function. This takes two arguments: the URL of the file you want to download, and the name of the file you want to save it as. They should be separated with a comma, and in quotes.

This is a *function*, a piece of code that does something. In this case, it downloads a file from the internet.

Please copy the code below and paste it into your new file.

```
download.file(
  "https://tierstatistik.identitas.ch/data/equids-topNamesFemale.csv",
  "input_data/equids-topNamesFemale.csv"
)

download.file(
  "https://tierstatistik.identitas.ch/data/equids-topNamesMale.csv",
  "input_data/equids-topNamesMale.csv"
)
```

### 1.5.3 Downloading directly from the web

If you're using RStudio on your own computer, you have the option to do this manually.

When we look at this data set, we can see that we have the option to **"Download the data (CSV)"**. Do that.

Now, find your downloaded file, and **put it in the same folder as your project.** I usually keep my raw, untouched data in a sub-folder called "input_data", but you can organize your files however you like.

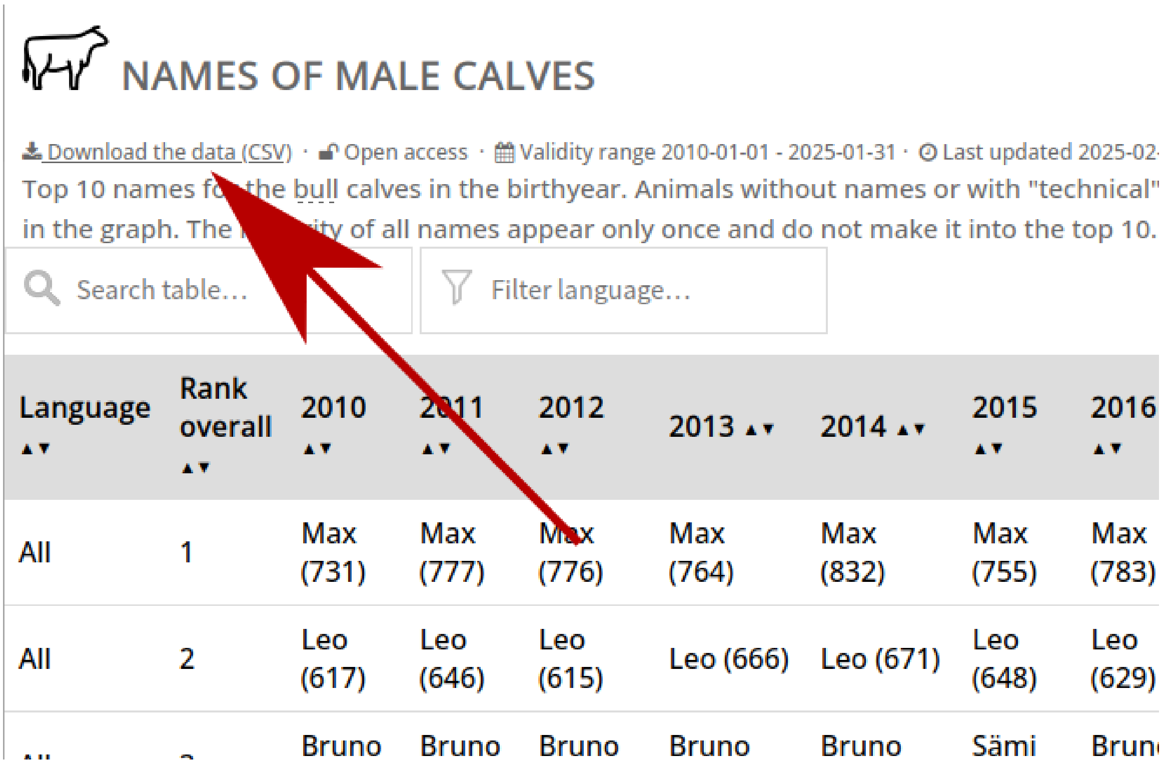



Figure 1.9: Download the data from the tierstatistik website

### 1.5.4 Importing data

The first thing we should always do with any data we get is to just to open it up and **take a look.** You should see it in your file screen, and if you click on it, you'll have the option to **View File**, which just opens it as a text file in RStudio.

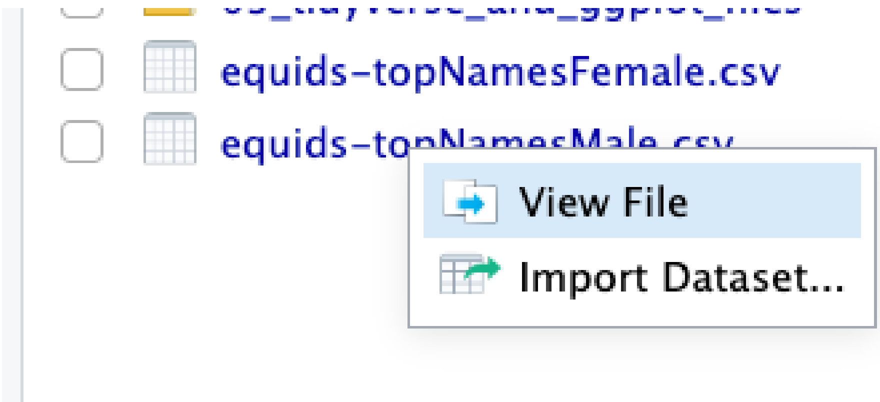



Figure 1.10: First, let's view the file.

You can also do this in something like Notepad if you prefer. It will look similar to this:

```
# Identitas AG. Validity: 2023-08-31. Evaluated: 2023-09-20
OwnerLanguage;Name;RankLanguage;CountLanguage;RankOverall;CountTotal
de;Luna;1;280;1;359
it;Luna;1;34;1;359
fr;Luna;2;45;1;359
it;Stella;2;26;2;159
de;Stella;4;104;2;159
fr;Stella;8;29;2;159
de;Fiona;2;114;3;153
it;Fiona;6;13;3;153
fr;Fiona;10;26;3;153
de;Cindy;2;114;4;131
```

Not very beautiful, but useful! Here are some things that we might notice:

1. The first line is some meta-information that we don't need. We don't want to import that.
2. This is in the form of a table. Each column of information is separated by a semicolon (;).
3. The second row is the names of each column of information. We should treat this row as a header.

Fortunately, RStudio has all the tools we need to help you do this. We can get started by opening the **"Import Dataset"** dialog.

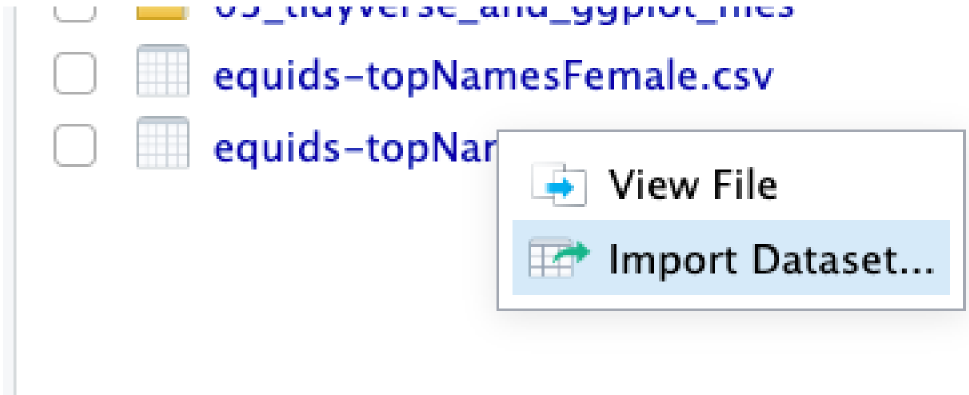


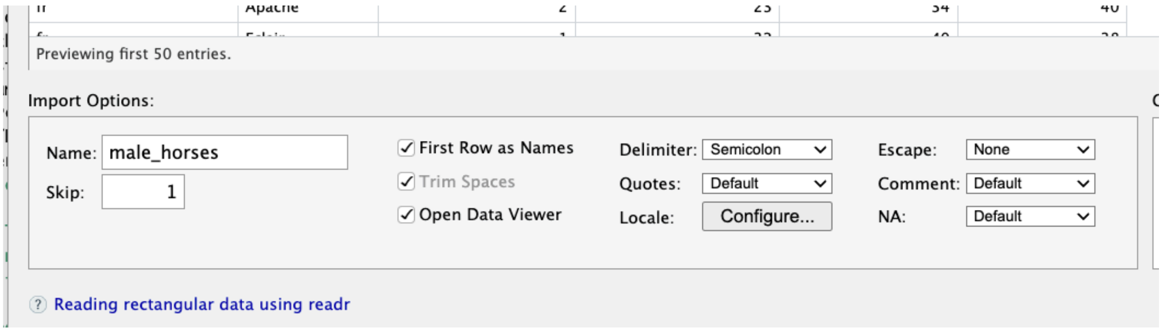

Figure 1.11: Now, we can import it into R.

In the bottom left, you'll see the import options. We'll need to adjust some of them to make this work.

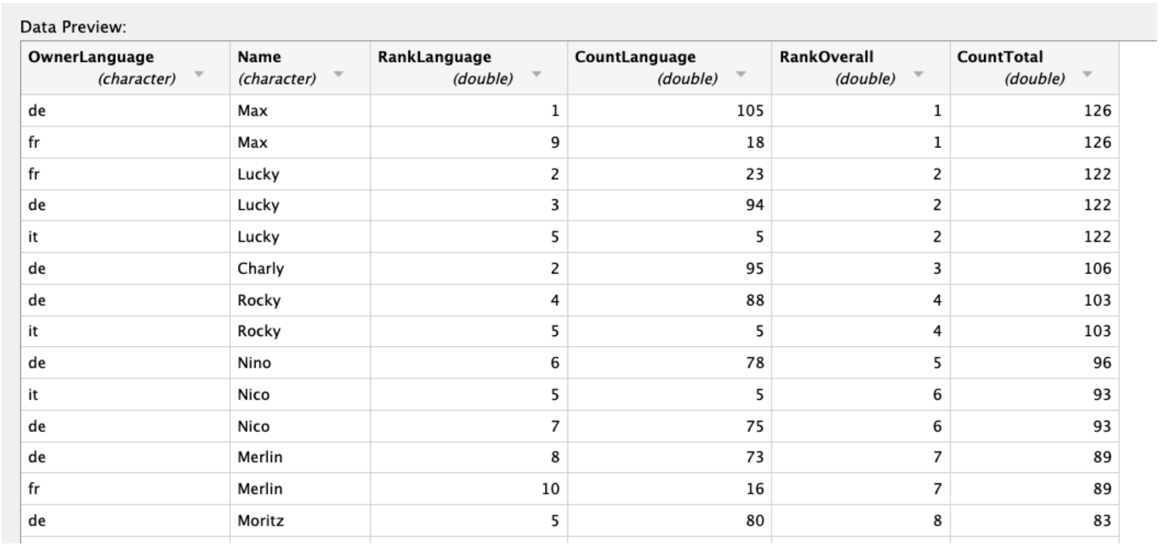
1. We should set **Skip:** to 1, to skip that first line of metadata.
2. The "Delimiter" is the thing that separates our data. For this dataset, we'll use a semicolon.
3. Make sure that "First Row as Names" is checked. This sets the column names.

4. “equids_topNamesMale” will be an annoying name to type. Change the **Name:** to something more convenient.

Your option box should look like this:




If you’ve done everything correctly, you should see that the columns have been cleanly separated, and each column has a name.



Data Preview:

| OwnerLanguage (character) | Name (character) | RankLanguage (double) | CountLanguage (double) | RankOverall (double) | CountTotal (double) |
|---|---|---|---|---|---|
| de | Max | 1 | 105 | 1 | 126 |
| fr | Max | 9 | 18 | 1 | 126 |
| fr | Lucky | 2 | 23 | 2 | 122 |
| de | Lucky | 3 | 94 | 2 | 122 |
| it | Lucky | 5 | 5 | 2 | 122 |
| de | Charly | 2 | 95 | 3 | 106 |
| de | Rocky | 4 | 88 | 4 | 103 |
| it | Rocky | 5 | 5 | 4 | 103 |
| de | Nino | 6 | 78 | 5 | 96 |
| it | Nico | 5 | 5 | 6 | 93 |
| de | Nico | 7 | 75 | 6 | 93 |
| de | Merlin | 8 | 73 | 7 | 89 |
| fr | Merlin | 10 | 16 | 7 | 89 |
| de | Moritz | 5 | 80 | 8 | 83 |

But don’t hit the import button! We want to focus on *reproducibility*, so we should make sure that our code runs without clicking these dialogues every time. Instead, copy the code from the code preview, and paste it into your source code.

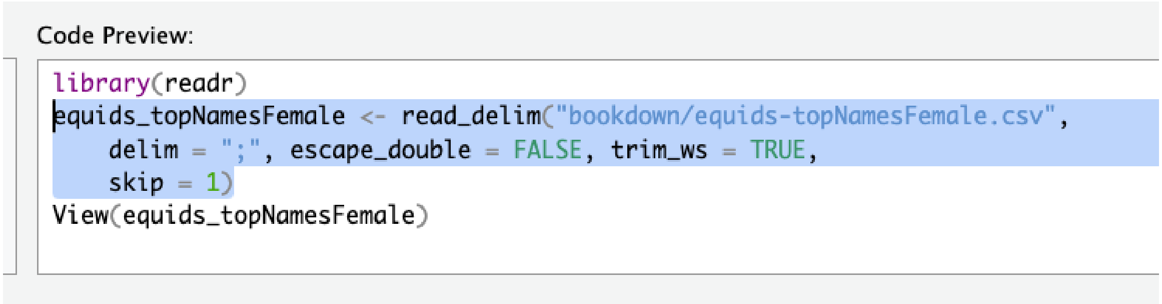



Figure 1.12: Copy and paste this into your main file.

Do the same thing with the other horse data set. After this step, your code should look something like this:

```
male_horses <- read_delim("input_data/equids-topNamesMale.csv",
    delim = ";", escape_double = FALSE, trim_ws = TRUE,
    skip = 1)

female_horses <- read_delim("input_data/equids-topNamesFemale.csv",
    delim = ";", escape_double = FALSE, trim_ws = TRUE,
    skip = 1)
```

Now, run this code. In the upper right hand panel on your screen, you should see that two new things have shown up, `male_horses` and `female_horses`.

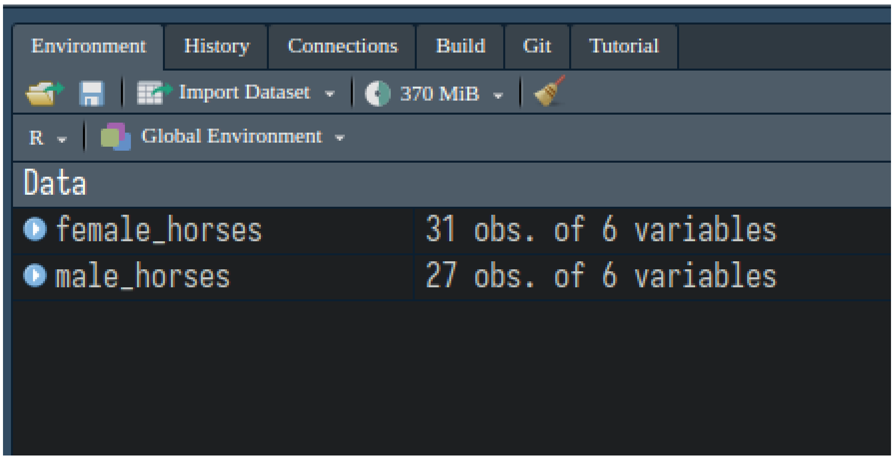



Figure 1.13: You should see these new variables in your environment.

This means that the data is now living in R's memory, and we can access it using that name.

Additionally, if you click on the name of the variable, you can see the data in a table format.

## 1.6 Data pipelines

But how to do things with this? This symbol will be your new best friend:

```
|>
```

This is called a **pipe**, and because it moves data from one place to the next.

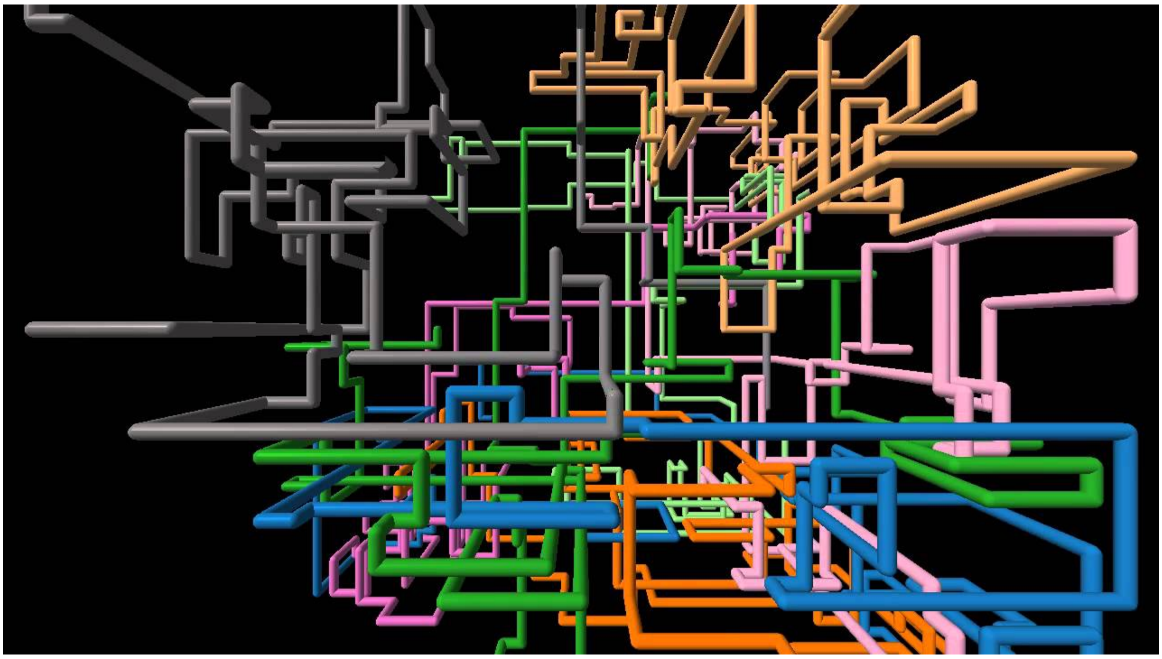


Figure 1.14: Pipes.

All this does is take the results of your last step, and pass it to your next function. This allows us to do many small steps at once, and split up our data pipelines into small, readable steps.

We combine these with specially-made functions to arrange and organize our data.

You'll be typing this a lot, and it's really handy to use a keyboard shortcut for the pipe. By default, it is something like Ctrl-Shift-M.

> **i What if it says `%>%` instead of `|>`?**
>
> When you type Ctrl-Shift-M, you might get a different result, namely this: `%>%`. To fix this, you can go to the go to the "Tools" > "Global Options" menu. Then open up the "Code" tab. Check the box that says "Use native pipe operator". You can find more details in the installation chapter.

### 1.6.1 Seeing the start of some data with `head()`

The first of these special functions we'll learn is **head()**, which just shows the first few lines of our data.

It's useful for taking a look at really big datasets; let's try it out with a pipe:

```
female_horses |>
  head()
```

```
# A tibble: 6 x 6
  OwnerLanguage Name   RankLanguage CountLanguage RankOverall CountTotal
  <chr>         <chr>         <dbl>         <dbl>       <dbl>      <dbl>
1 de            Luna              1           276           1        348
2 it            Luna              1            29           1        348
3 fr            Luna              2            43           1        348
4 it            Stella            2            27           2        153
5 de            Stella            3            97           2        153
6 fr            Stella            5            29           2        153
```

### 1.6.2 Counting rows with `count()`

Another useful function is count(), which gives the total number of rows, divided by the number of columns you select. For example, if I wanted to know the total number of names in each language, I could pipe `|>` the OwnerLanguage into count.

The output is always the input columns and `n`, which is the number of rows.

```
female_horses |>
  count(OwnerLanguage)
```

```
# A tibble: 3 x 2
  OwnerLanguage     n
  <chr>         <int>
1 de               10
2 fr               13
3 it               10
```

### 1.6.3 Filtering out only the stuff we want with `filter()`

This gives us a little preview of what we're looking at, so new we can go ahead and search for the data that we want, by filtering out data that we don't need.

One of the most important of these functions is **filter()**. Filtering only keeps the rows that we want, just like a coffee filter keeps only the liquid we want to drink, while getting rid of the gritty ground beans.

Let's say we wanted to find out the most common name for a horse with a German-speaking owner. Having looked at our data's head, we can see that the column OwnerLanguage will tell us this information. To keep only the German data, we could use a filter like this:

```r
female_horses |>
  filter(OwnerLanguage == "de")
```

```
# A tibble: 10 x 6
   OwnerLanguage Name   RankLanguage CountLanguage RankOverall CountTotal
   <chr>         <chr>         <dbl>         <dbl>       <dbl>      <dbl>
 1 de            Luna              1           276           1        348
 2 de            Stella            3            97           2        153
 3 de            Fiona             2           114           3        152
 4 de            Cindy             4            96           5        112
 5 de            Lisa              6            87           6        110
 6 de            Fanny             9            79           7        105
 7 de            Nora              7            84           8        104
 8 de            Sina              5            92           9         99
 9 de            Lara              9            79          10         98
10 de            Ronja             8            80          14         83
```

That's great, but what if we only wanted the top 3 names? We could use a second filter, with one piping into the next.

```r
female_horses |>
  filter(OwnerLanguage == "de") |>
  filter(RankLanguage <= 3)
```

```
# A tibble: 3 x 6
  OwnerLanguage Name   RankLanguage CountLanguage RankOverall CountTotal
  <chr>         <chr>         <dbl>         <dbl>       <dbl>      <dbl>
1 de            Luna              1           276           1        348
2 de            Stella            3            97           2        153
3 de            Fiona             2           114           3        152
```

Even better, but those maybe we don't need those other columns in our final analysis, so we can just select the ones that we need.

### 1.6.4 Selecting only the columns we want with `select()`

Just like **filter()** filters out the rows that we want, **select()** can select only the columns that we want. We simply pass the names of the columns that we need, and only those will be taken.

In this example, we just want the top 3 horses, as well as the language count.

```
female_horses |>
  filter(OwnerLanguage == "de") |>
  filter(RankLanguage <= 3) |>
  select(Name, CountLanguage)
```

```
# A tibble: 3 x 2
  Name   CountLanguage
  <chr>          <dbl>
1 Luna             276
2 Stella            97
3 Fiona            114
```

Great! But CountLanguage is kind of an awkward name. Can we rename it?

### 1.6.5 Giving columns better names with `rename()`

The rename function, as the name implies, renames columns. We can use it to make our data more readable.

```
female_horses |>
  filter(OwnerLanguage == "de") |>
  filter(RankLanguage <= 3) |>
  select(Name, CountLanguage) |>
  rename(Count = CountLanguage)
```

```
# A tibble: 3 x 2
  Name   Count
  <chr>  <dbl>
1 Luna     276
2 Stella    97
3 Fiona    114
```

### 1.6.6 Changing data with `mutate()`

Much cleaner. However, sometimes we want to change something inside the cell. We can use **mutate()** to make new columns with slightly changed data, or replace a column that we have, using a function.

Maybe we want to make the names uppercase, like we are yelling at our horse. We already know that you can use **toupper** to change some text, like so:

```
toupper("yelling")
```

```
[1] "YELLING"
```

To apply this to our data, we can use the **mutate()**, like so:

```
female_horses |>
  filter(OwnerLanguage == "de") |>
  filter(RankLanguage <= 3) |>
  select(Name, CountLanguage) |>
  rename(Count = CountLanguage) |>
  mutate(loud_name = toupper(Name))
```

```
# A tibble: 3 x 3
  Name   Count loud_name
  <chr>  <dbl> <chr>
1 Luna     276 LUNA
2 Stella    97 STELLA
3 Fiona    114 FIONA
```

We can even overwrite the original column with mutate, instead of making a new one:

```
female_horses |>
  filter(OwnerLanguage == "de") |>
  filter(RankLanguage <= 3) |>
  select(Name, CountLanguage) |>
  rename(Count = CountLanguage) |>
  mutate(Name = toupper(Name))
```

```
# A tibble: 3 x 2
  Name   Count
  <chr>  <dbl>
1 LUNA     276
2 STELLA    97
3 FIONA    114
```

Soon, we will combine this with the male dataset, but we need to remember if each of these names is for a mare or a stallion. We can simply mutate a new column with the sex of the horse.

```
female_horses |>
  filter(OwnerLanguage == "de") |>
  filter(RankLanguage <= 3) |>
  select(Name, CountLanguage) |>
  rename(Count = CountLanguage) |>
  mutate(Name = toupper(Name)) |>
  mutate(Sex = "F")
```

```
# A tibble: 3 x 3
  Name   Count Sex
  <chr>  <dbl> <chr>
1 LUNA     276 F
2 STELLA    97 F
3 FIONA    114 F
```

Pretty clean, I'm happy with that!

### 1.6.7 Saving variables with <-

Until now, we've been doing things to our data, then just printing it out. Sometimes, we want to save our work so we can use it later. We can do this with the **assignment operator**, `<-`.

The assignment operator saves whatever we put in front of it to memory, so we can use it later. For example, if we just wanted to save our name, we could do this:

```
my_name <- "Morley"
```

When we run this code, nothing will show up in the console. However, if we type `my_name` and run it, it will print out your name. You should also see it listed in the environment panel in the upper right hand corner.

In this case, we want to use the assignment operator to save our new data frame. We can call it german_mares. We put this at the beginning of the line, and then run the code.

```
german_mares <- female_horses |>
  filter(OwnerLanguage == "de") |>
  filter(RankLanguage <= 3) |>
  select(Name, CountLanguage) |>
  rename(Count = CountLanguage) |>
  mutate(Name = toupper(Name))  |>
  mutate(Sex = "F")
```

Note that nothing shows up when you type this, because we haven't told R to show it to us. If you're feeling paranoid, just type the name of a variable and it will print.

```
german_mares
```

```
# A tibble: 3 x 3
  Name   Count Sex  
  <chr>  <dbl> <chr>
1 LUNA     276 F    
2 STELLA    97 F    
3 FIONA    114 F    
```

> **What if I wanted to call it something else?**
>
> You can name your variables whatever you like, but there are some rules.
>
> - You can't have spaces in the name.
> - You can't start with a number.
> - Capitalization matters, so `my_name` is different from `My_Name`.
> - You can't use special characters like `!`, `@`, `#`, `$` or `%`.
> - Two things can't have the same name, so you can't have two variables called `my_name`.
>
> In addition, we have some general conventions in R:
>
> - We use lowercase for variable names, and separate words with an underscore, like `my_name`.
> - The name should be descriptive, so you know what it is later.

### 1.6.8 Combining data with `bind_rows()`

The mares are ready, but what about the stallions? With a little copy-paste, we can simply re-do the same process for the males:

```
german_stallions <- male_horses |> # This line is different.
  filter(OwnerLanguage == "de") |>
  filter(RankLanguage <= 3) |>
  select(Name, CountLanguage) |>
  rename(Count = CountLanguage) |>
  mutate(Name = toupper(Name))  |>
  mutate(Sex = "M") # This line is different.

german_stallions
```

```
# A tibble: 3 x 3
  Name   Count Sex
  <chr>  <dbl> <chr>
1 MAX      101 M
2 LUCKY     89 M
3 CHARLY    89 M
```

Our next step is to combine the two datasets into one. We can do this with **bind_rows()**. It adds one dataset to another, vertically. We pass the second dataset as an argument, and it plops them one on top of another.

```
all_horses <- german_mares |>
  bind_rows(german_stallions)

all_horses
```

```
# A tibble: 6 x 3
  Name   Count Sex
  <chr>  <dbl> <chr>
1 LUNA     276 F
2 STELLA    97 F
3 FIONA    114 F
4 MAX      101 M
5 LUCKY     89 M
6 CHARLY    89 M
```

### 1.6.9 bind_cols()

A related function is **bind_cols()**, which adds one dataset to another, horizontally. This is useful when you have two datasets with the same number of rows, and you want to combine them into one dataset with more columns.

Here is an example of using `bind_rows()` and `bind_cols()` to combine some datasets:

First, I'm going to make some fake data, in this case a table with people's names, ages and jobs.

```
some_people <- tibble(
  name = c("Urs", "Karl", "Hans"),
  age = c(23, 45, 67),
  job = c("data scientist", "electrician", "artist")
)

some_people
```

```
# A tibble: 3 x 3
  name    age job
  <chr> <dbl> <chr>
1 Urs      23 data scientist
2 Karl     45 electrician
3 Hans     67 artist
```

Now I'm going to make another table with some other people's names, ages and jobs.

```
other_people <- tibble(
  name = c("Heidi", "Ursula", "Gretel"),
  age = c(34, 56, 78),
  job = c("engineer", "doctor", "PhD student")
)

other_people
```

```
# A tibble: 3 x 3
  name     age job
  <chr>  <dbl> <chr>
1 Heidi     34 engineer
2 Ursula    56 doctor
3 Gretel    78 PhD student
```

if we want to put them together on top of each other, we use `bind_rows()`. Note that they have to have the **same column names.**

```
some_people |>
  bind_rows(other_people)
```

```
# A tibble: 6 x 3
  name     age job
  <chr>  <dbl> <chr>
1 Urs       23 data scientist
2 Karl      45 electrician
3 Hans      67 artist
4 Heidi     34 engineer
5 Ursula    56 doctor
6 Gretel    78 PhD student
```

I can then assign them to a new variable, `all_people`. This saves it in our environment.

```r
all_people <- some_people |>
  bind_rows(other_people)
```

However, I not have some other information about these people, such as their height and weight. I want to add this to the `all_people` dataset.

```r
other_information <- tibble(
  height = c(180, 160, 170, 220, 190, 160),
  weight = c(80, 70, 90, 120, 90, 60)
)
```

I can do this with `bind_cols()`. Note that the two tables must have the **same number of rows.**

```r
all_people |>
  bind_cols(other_information)
```

```
# A tibble: 6 x 5
  name     age job            height weight
  <chr>  <dbl> <chr>           <dbl>  <dbl>
1 Urs       23 data scientist    180     80
2 Karl      45 electrician       160     70
3 Hans      67 artist            170     90
4 Heidi     34 engineer          220    120
5 Ursula    56 doctor            190     90
6 Gretel    78 PhD student       160     60
```

## 1.7 Check your knowledge

1. Find the location of the folder for the project you made. Hopefully you put it somewhere sensible like your "documents" folder.
2. Review how to use the following functions:
    1. `head()`
    2. `tail()`
    3. `count()`
    4. `filter()`
    5. `select()`
    6. `rename()`
    7. `mutate()`
    8. `bind_rows()`
    9. `bind_cols()`
3. Know that `<-` does, and how this is different from printing to the console.

## 1.8 Homework: Cleaning a similar dataset

The dataset for cattle is arranged differently. Can you figure out how to produce the same final table for German cows?

The raw data can be found here:

https://tierstatistik.identitas.ch/en/cattle-NamesFemaleCalves.html

https://tierstatistik.identitas.ch/en/cattle-NamesMaleCalves.html

Hint: There are a lot of different years in this data set. We only need the current year.

Please save your script as `week_1_homework_(your_name).R` and email it to me by Tuesday, February 25th.

If you're using the web server, you'll want to find the file in the lower right hand corner, check the box next to the file, then go to (gear symbol) > "Export ...". This will download the file to your computer.

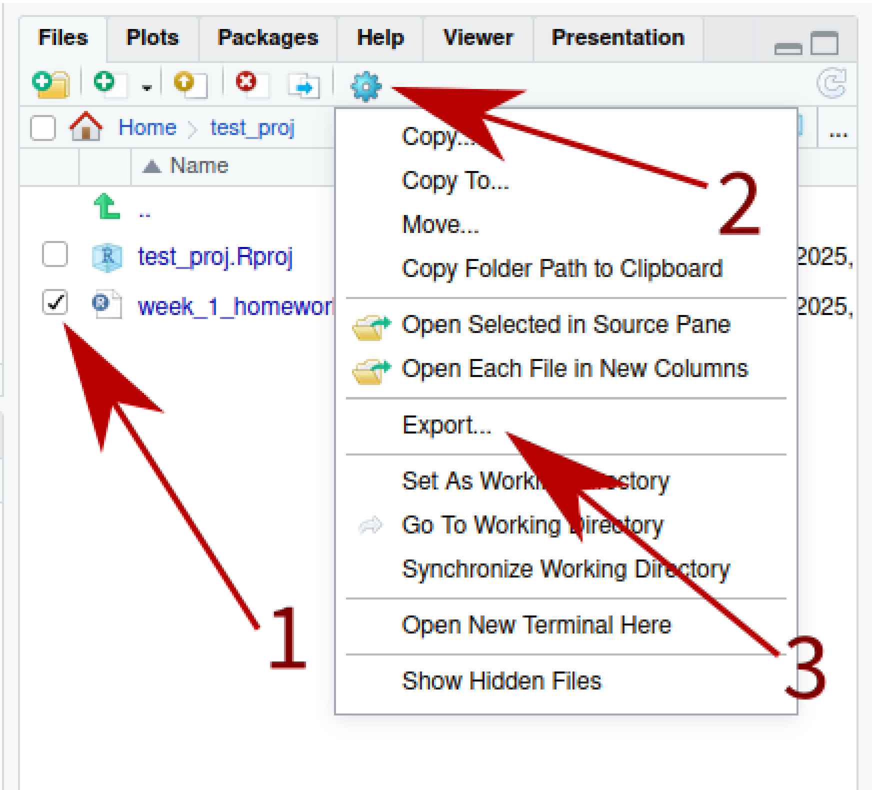



Figure 1.15: These are the steps to download your stuff.

# 2 Tidyverse 2: Data transformation

## 2.1 Downloading data

This week, we're going to look at three different common data formats in R. You'll find these all the time when you search for data online.

Rather than downloading the files manually, we're going to use R to download them for us. This is a good way to automate the process, and also makes it easier to share the code with others.

For this, we'll use the `download.file()` function. This takes two arguments:

1. The URL of the *file* to download
2. What you want the file to be named on your computer

Make sure the file type matches the file type you're downloading. If you're downloading a .jpg file, the file name should end in .jpg.

For example, if we wanted to download a picture from Wikipedia, we could use:

```
download.file(
  "https://upload.wikimedia.org/wikipedia/commons/d/d8/Panthera_tigris_corbetti_(Tierpark_
  "picture_of_a_tiger.jpg"
)
```

Then you'll have a cool picture of a tiger on your computer.

## 2.2 CSV

First, we're going to look at a CSV file. CSV stands for "comma-separated values". These are two-dimensional tables, where each row is a line in the file, and each column is separated by a comma. Here's an example:

```
"name","age","married"
"Gunther",42,TRUE
"Gerhard",38,TRUE
"Heidi",29,FALSE
```

This evaluates to a simple table, like this:

```
     name age married
1 Gunther  42    TRUE
2 Gerhard  38    TRUE
3   Heidi  29   FALSE
```

These are great because you can open them in a text editor and read them, and are simple enough to edit. They're also easy to read into R.

Despite the name, CSV files don't always use commas to separate the columns. Sometimes they use semicolons, or tabs, or other characters; the Swiss government really likes semicolons for some reason.

Let's take a look at a real-world example. We're going to use the Swiss government's Bundesamt für Statistik (BFS) website to download some data, about incomes for every commune in Switzerland, originally from here:

https://www.atlas.bfs.admin.ch/maps/13/de/15830_9164_8282_8281/24776.html

We find the download link, and use `download.file()` to download it:

```
download.file(
  "https://www.atlas.bfs.admin.ch/core/projects/13/xshared/csv/24776_131.csv",
  "input_data/income.csv"
)
```

Once again, let's take a look at the raw data. Open it in a text editor, and it should look something like this:

```
"GEO_ID";"GEO_NAME";"VARIABLE";VALUE;"UNIT";"STATUS";"STATUS_DESC";"DESC_VAL";"PERIOD_REF"
"1";"Aeugst am Albis";"Steuerbares Einkommen, in Mio. Franken";98;"Franken";"A";"Normaler
"1";"Aeugst am Albis";"Steuerbares Einkommen pro Einwohner/-in, in Franken";50443;"Franken
"2";"Affoltern am Albis";"Steuerbares Einkommen, in Mio. Franken";391;"Franken";"A";"Norma
```

We can see the following:

1. The data is once again separated by semicolons.
2. This has no metadata row, the first row is the header.

In the same way as we did last week, we can use **Import Dataset** to import the data into RStudio. You can see complete instructions in the previous chapter. The code that we get back should look something like this:

```
income_per_gemeinde <- read_delim("input_data/income.csv",
  delim = ";", escape_double = FALSE, trim_ws = TRUE
)
```

Another option is to use use read_delim() on the URL itself. This reads the data directly from the URL, without downloading it to your computer:

```
income_per_gemeinde <- read_delim("https://www.atlas.bfs.admin.ch/core/projects/13/xshared
```

This can be a little dangerous, however, as the data might change, or the website could go down, and your data is lost forever.

### 2.2.1 Looking at the data with `glimpse()`

This data has a lot of columns, and isn't always the easiest to read. One convenient way to glimpse at the data is the **glimpse()** function, which shows us the first few rows of each column:

```
income_per_gemeinde |> glimpse()
```

```
Rows: 4,510
Columns: 16
$ GEO_ID      <dbl> 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, ~
$ GEO_NAME    <chr> "Aeugst am Albis", "Aeugst am Albis", "Affoltern am Albis"~
$ VARIABLE    <chr> "Steuerbares Einkommen, in Mio. Franken", "Steuerbares Ein~
$ VALUE       <dbl> 98, 50443, 391, 32180, 224, 40564, 148, 40398, 155, 41909,~
$ UNIT        <chr> "Franken", "Franken", "Franken", "Franken", "Franken", "Fr~
$ STATUS      <chr> "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A"~
$ STATUS_DESC <chr> "Normaler Wert", "Normaler Wert", "Normaler Wert", "Normal~
$ DESC_VAL    <lgl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
$ PERIOD_REF  <chr> "2017-01-01/2017-12-31", "2017-01-01/2017-12-31", "2017-01~
$ SOURCE      <chr> "ESTV", "ESTV", "ESTV", "ESTV", "ESTV", "ESTV", "ESTV", "E~
$ LAST_UPDATE <date> 2021-01-07, 2021-01-07, 2021-01-07, 2021-01-07, 2021-01-0~
$ GEOM_CODE   <chr> "polg", "polg", "polg", "polg", "polg", "polg", "polg", "p~
$ GEOM        <chr> "Politische Gemeinden", "Politische Gemeinden", "Politisch~
$ GEOM_PERIOD <date> 2017-01-01, 2017-01-01, 2017-01-01, 2017-01-01, 2017-01-0~
$ MAP_ID      <dbl> 24776, 24776, 24776, 24776, 24776, 24776, 24776, 24776, 24~
$ MAP_URL     <chr> "https://www.atlas.bfs.admin.ch/maps/13/map/mapIdOnly/2477~
```

This flips the data frame on its side, so that the columns are now rows, and the rows are now columns. This makes it easier to see the data types, but is really only useful for taking a peek at our data.

For this example, we'll want the GEO_NAME, VARIABLE, and VALUE columns. We can use the **select()** function to select only those columns:

```
income_per_gemeinde <- income_per_gemeinde |>
  select(GEO_NAME, VARIABLE, VALUE)
```

We can now easily look at the data that we're interested in:

```
income_per_gemeinde |> head()
```

```
# A tibble: 6 x 3
  GEO_NAME           VARIABLE                                             VALUE
  <chr>              <chr>                                                <dbl>
1 Aeugst am Albis    Steuerbares Einkommen, in Mio. Franken                  98
2 Aeugst am Albis    Steuerbares Einkommen pro Einwohner/-in, in Franken 50443
3 Affoltern am Albis Steuerbares Einkommen, in Mio. Franken                 391
4 Affoltern am Albis Steuerbares Einkommen pro Einwohner/-in, in Franken 32180
5 Bonstetten         Steuerbares Einkommen, in Mio. Franken                 224
6 Bonstetten         Steuerbares Einkommen pro Einwohner/-in, in Franken 40564
```

## 2.3 Pivoting data with `pivot_wider()`

However, we can see this data still has a pretty big problem: the VARIABLE column contains the name of the variable, and the VALUE column contains the value of the variable. This means that the VALUE column actually represents two things at the same time: The total income of the commune, and the per-capita income of the commune.

This is a common problem in data analysis. Recalling Wickham's paper, we want every column to represent a single variable, and every row to represent a single observation, which he calls "tidy data".

We can fix this by using the **pivot_wider()** function, which takes the values in one column, and turns them into columns. We'll use the VARIABLE column as the column names, and the VALUE column as the values. To do this, we'll use two arguments for `pivot_wider()`: `names_from`, which is the column that we want to use as the column names, and `values_from`, which is the column that we want to use as the values.

```
income_per_gemeinde <- income_per_gemeinde |>
  pivot_wider(names_from = VARIABLE, values_from = VALUE)
```

This can be hard to get your brain around, so let's take a look at the data before and after:

### 2.3.1 Before

```
# A tibble: 4,510 x 3
   GEO_NAME           VARIABLE                          VALUE
   <chr>              <chr>                             <dbl>
 1 Aeugst am Albis    Steuerbares Einkommen, in Mio.       98
 2 Aeugst am Albis    Steuerbares Einkommen pro Einw 50443
 3 Affoltern am Albis Steuerbares Einkommen, in Mio.      391
 4 Affoltern am Albis Steuerbares Einkommen pro Einw 32180
 5 Bonstetten         Steuerbares Einkommen, in Mio.      224
 6 Bonstetten         Steuerbares Einkommen pro Einw 40564
 7 Hausen am Albis    Steuerbares Einkommen, in Mio.      148
 8 Hausen am Albis    Steuerbares Einkommen pro Einw 40398
 9 Hedingen           Steuerbares Einkommen, in Mio.      155
10 Hedingen           Steuerbares Einkommen pro Einw 41909
# i 4,500 more rows
```

### 2.3.2 After

```
# A tibble: 2,255 x 3
   GEO_NAME           `St. Eink, in Mio.` `St. Eink. pro Einw`
   <chr>                            <dbl>                <dbl>
 1 Aeugst am Albis                     98                50443
 2 Affoltern am Albis                 391                32180
 3 Bonstetten                         224                40564
 4 Hausen am Albis                    148                40398
 5 Hedingen                           155                41909
 6 Kappel am Albis                     50                44353
 7 Knonau                             84                36395
 8 Maschwanden                         20                31437
 9 Mettmenstetten                     200                41023
10 Obfelden                           177                33089
# i 2,245 more rows
```

(Column names were abbreviated to fit on the screen)

The opposite of `pivot_wider()` is `pivot_longer()`, which takes columns and turns them into rows. You can really only understand this from practice, so you'll get more exposure to it next week.

## 2.4 Renaming columns with `colnames()` and `rename()`

This data is now in the shape we want it, but the column names are still an absolute mess. I really don't want to type `Steuerbares Einkommen pro Einwohner/-in, in Franken` every time I want to refer to the per-capita income column. We can rename all the columns by just assigning a vector of names to the `colnames()` function:

```
colnames(income_per_gemeinde) <- c("name", "total_income", "per_capita_income")
income_per_gemeinde |> head()
```

```
# A tibble: 6 x 3
  name                total_income per_capita_income
  <chr>                      <dbl>             <dbl>
1 Aeugst am Albis               98             50443
2 Affoltern am Albis           391             32180
3 Bonstetten                   224             40564
4 Hausen am Albis              148             40398
5 Hedingen                     155             41909
6 Kappel am Albis               50             44353
```

Note that if we only wanted to rename one column, it might be easier to use the **rename()** function:

With the `rename()` function, remember that the new name comes first, and the old name comes second.

```
income_per_gemeinde <- income_per_gemeinde |>
  rename(gemeinde_name = name)
income_per_gemeinde |> head()
```

```
# A tibble: 6 x 3
  gemeinde_name       total_income per_capita_income
  <chr>                      <dbl>             <dbl>
1 Aeugst am Albis               98             50443
2 Affoltern am Albis           391             32180
3 Bonstetten                   224             40564
4 Hausen am Albis              148             40398
5 Hedingen                     155             41909
6 Kappel am Albis               50             44353
```

## 2.5 Math on columns.

A little housecleaning: The total income is in millions of francs, so we'll multiply it by 1,000,000 to get the actual value. This will save some confusion later on.

To change a column, we can just assign a new value to it using `mutate()`:

```
income_per_gemeinde <- income_per_gemeinde |>
  mutate(total_income = total_income * 1e6)
```

## 2.6 Sorting data with `arrange()`

We can sort the data by using the **arrange()** function. This takes the column that we want to sort by, and the direction that we want to sort in. We can use `desc()` to sort in descending order, or `asc()` to sort in ascending order. For example, to sort by per-capita income, we can use:

```
income_per_gemeinde <- income_per_gemeinde |>
  arrange(desc(per_capita_income))

income_per_gemeinde |> head(10)
```

```
# A tibble: 10 x 3
   gemeinde_name   total_income per_capita_income
   <chr>                  <dbl>             <dbl>
 1 Vaux-sur-Morges     65000000            324181
 2 Mies               334000000            162965
 3 Anières            388000000            158061
 4 Feusisberg         824000000            156325
 5 Wollerau           985000000            138662
 6 Crésuz              47000000            137880
 7 Cologny            587000000            106112
 8 Montricher         104000000            105971
 9 Buchillon           67000000            104523
10 Vandoeuvres        258000000            103113
```

This gives us the 10 communes with the highest per-capita income.

## 2.7 Class Work: Getting data from a data frame

Use this data set to answer the following questions:

1. Which is the poorest commune in Switzerland, on a per-capita basis?
2. Which commune in Switzerland has the highest total income?
3. Can you use these two columns to figure out the population of each commune?

## 2.8 JSON

Our next data format is JSON. JSON stands for "JavaScript Object Notation", as it was originally designed to be used in JavaScript. It's a very flexible format, and is used in pretty much every programming language.

Let's download and take a look at some JSON, originally from here:

This is a list of names given to babies in Basel, by year. We can download it using:

```
download.file(
  "https://data.bs.ch/api/v2/catalog/datasets/100192/exports/json",
  "input_data/basel_babies.json"
)
```

When we look at the raw data, we can see that it's a list of key-value pairs, where the keys are the column names, and the values are the values. This is a very flexible format, and can be used to represent pretty much any data structure. This is a huge dataset

```
[{"jahr": "2012", "geschlecht": "M", "vorname": "Jacob", "anzahl": 1},
 {"jahr": "2012", "geschlecht": "W", "vorname": "Ja\u00ebl", "anzahl": 1},
 {"jahr": "2012", "geschlecht": "M", "vorname": "Jai", "anzahl": 1},
 ...
 ...
 ...
 {"jahr": "2019", "geschlecht": "W", "vorname": "Tara", "anzahl": 2},
 {"jahr": "2019", "geschlecht": "W", "vorname": "Tatjana", "anzahl": 1},
 {"jahr": "2019", "geschlecht": "W", "vorname": "Tenzin", "anzahl": 1}
]
```

However, R doesn't really have the ability to read JSON on it's own, so we'll need to use a package to read it. We'll use the **jsonlite** package, which has a function called **read_json()** that reads JSON files into R. Install and load the library in the usual way:

```
install.packages("jsonlite")
```

```
library(jsonlite)
```

Now you can use the function `read_json()` to read the file into R like so:

```
basel_babies <-
  read_json("input_data/basel_babies.json", simplifyVector = TRUE)
```

`simplifyVector` is a parameter that tells R to simplify the data structure, assuming that it is in a tabular format. You'll almost always want to use this option, unless you're working with a very complex JSON file.

Let's look at the result:

```
basel_babies |> head()
```

```
  jahr geschlecht vorname anzahl
1 2012          W   Neyla      2
2 2012          M Niccolo      1
3 2012          W   Nikki      1
4 2012          M  Nikola      2
5 2012          M    Nils      3
6 2012          W    Nina      3
```

As an English-language class, let's rename the columns to English:

```
basel_babies <- basel_babies |>
  rename(
    name = vorname,
    year = jahr,
    sex = geschlecht,
    total = anzahl,
  )

basel_babies |> head()
```

```
  year sex    name total
1 2012   W   Neyla     2
2 2012   M Niccolo     1
3 2012   W   Nikki     1
4 2012   M  Nikola     2
5 2012   M    Nils     3
6 2012   W    Nina     3
```

## 2.9 Group_by and Summarize

This is a pretty big data set! We can see the number of rows using the **nrow()** function:

```
nrow(basel_babies)
```

```
[1] 23676
```

That's a lot of babies. But sometimes we need to condense this information into a single number.

For this, we can use the `group_by()` and `summarize()` [1] functions. These are a little tricky to understand, so let's take a look at an example. Let's say we want to know how many babies were born in Basel per year. We can use `group_by()` to group the data by year, and then `summarize()` to summarize the data.

```
basel_babies |>
  group_by(year) |>
  summarise(total_by_year = sum(total))
```

```
# A tibble: 19 x 2
   year  total_by_year
   <chr>         <int>
 1 2006           1662
 2 2007           1667
 3 2008           1695
 4 2009           1775
 5 2010           1910
 6 2011           1868
 7 2012           1930
 8 2013           1962
 9 2014           1957
10 2015           2065
11 2016           2172
12 2017           2083
13 2018           2079
14 2019           2067
15 2020           2000
16 2021           2066
17 2022           1791
18 2023           1878
19 2024           1656
```

We first grouped the data by year, and then summarized the data by summing the total column. You can use quite a few different functions in `summarize()`, including `sum()`, `mean()`, `median()`, `min()`, `max()`, and many more.

## 2.10 RDS and friends

.RDS files are a special format that R uses to save data. They're a *binary format*, so you can't open them in a text editor, but they're very fast to read and write. They're also very

[1] R is friendly to both Brits and Americans, so it has both the `summarise()` and `summarize()` functions, which do the exact same thing.

easy to use, because they save all the metadata about the data frame, including the column names, data types, and more.

These are often used for your intermediary data sets, to just save something quickly and share it with a colleague. you can simply write them with the `write_rds()` function:

```
basel_babies |> write_rds("babies.rds")
```

Likewise, you can read them with the `read_rds()` function:

```
read_rds("babies.rds")
```

**However**, there are two problems with RDS files:

1. They only work in R. If you want to share your data with a colleague who uses Python, they're out of luck.
2. They're not human-readable. If you want to take a peek at the data, you can't just open it in a text editor.

## 2.11 Class work: Grouping and summarizing

Let's say we want to know how many Basel babies have names for each letter of the alphabet.

1. Use `mutate()` to make a new column with the first letter of each name. One function you can use inside mutate is `str_sub()`. `str_sub()` takes a string, and returns a part of that string. For example, `str_sub("hello", 1, 4)` returns "hell", from the first to the 4th letters of hello.

2. Use `group_by()` and `summarize()` to count the number of babies with each first letter.

3: Bonus: download the package `stringi`, which has the function `stri_trans_general()`. Look up how it works using `?stri_trans_general`. Use this to get rid of all the ãccènt märks ìn thë nàmĕs.

Your resulting table should look like this:

```
# A tibble: 10 x 2
   first_letter total
   <chr>        <int>
 1 A             4907
 2 B              816
 3 C             1133
 4 D             1276
 5 E             3200
```

```
 6 F                 962
 7 G                 766
 8 H                 774
 9 I                 778
10 J                2349
```

## 2.12 XLSX

Our last data format for the day is XLSX. This is a proprietary format, and is used by Microsoft Excel. I'd discourage your form using this unless you have to, but sometimes you'll find it in the wild, and you might have less gifted colleagues who insist on using it.

Let's download and take a look at some XLSX data, originally from the US Census Bureau:

```
download.file(
  "https://www2.census.gov/programs-surveys/decennial/2020/data/apportionment/apportionmen
  "input_data/state_population.xlsx"
)
```

Of course, you can always open them in Excel, but that's not very reproducible. Instead, we'll use the **readxl** package to read the data into R.

Load the library in the usual way:

```
library(readxl)
```

Now, you can click on your downloaded file in the file editor, and import it just like you did with the CSV file. You can see complete instructions in the last chapter.

The code that we get back should look something like this:

```
state_population <- read_excel("input_data/state_population.xlsx",
  skip = 3
)
```

Let's take a look at the data frame we get back:

```
state_population |> head()
```

```
# A tibble: 6 x 3
  AREA       `RESIDENT POPULATION (APRIL 1, 2020)` This cell is intentionally ~1
  <chr>                                      <dbl> <lgl>
1 Alabama                                  5024279 NA
2 Alaska                                    733391 NA
```

```
3 Arizona                                    7151502 NA
4 Arkansas                                   3011524 NA
5 California                                39538223 NA
6 Colorado                                   5773714 NA
# i abbreviated name: 1: `This cell is intentionally blank.`
```

We have three columns:

1. AREA
2. RESIDENT POPULATION (APRIL 1, 2020)
3. This cell is intentionally blank.

First, let's rename the columns to something a little more sensible:

```r
colnames(state_population) <- c("state_or_territory", "population", "blank")
```

Next, we can get rid of the `blank` column. A quick way to do this is to use the `select()` function with a minus sign in front of the column name that we don't want:

```r
state_population <- state_population |>
  select(-blank)

state_population
```

```
# A tibble: 55 x 2
   state_or_territory   population
   <chr>                     <dbl>
 1 Alabama                 5024279
 2 Alaska                   733391
 3 Arizona                 7151502
 4 Arkansas                3011524
 5 California             39538223
 6 Colorado                5773714
 7 Connecticut             3605944
 8 Delaware                 989948
 9 District of Columbia     689545
10 Florida                21538187
# i 45 more rows
```

When we look at the data frame, we can see that the last few rows should be removed, but maybe Puerto Rico should be included in our calculations. [2]

[2] https://en.wikipedia.org/wiki/Political_status_of_Puerto_Rico

```
# A tibble: 10 x 2
   state_or_territory                                                   population
   <chr>                                                                     <dbl>
 1 "Vermont"                                                                643077
 2 "Virginia"                                                              8631393
 3 "Washington"                                                            7705281
 4 "West Virginia"                                                         1793716
 5 "Wisconsin"                                                             5893718
 6 "Wyoming"                                                                576851
 7 "TOTAL RESIDENT POPULATION1"                                          331449281
 8 "Puerto Rico"                                                           3285874
 9 "TOTAL RESIDENT POPULATION, INCLUDING PUERTO RICO"                    334735155
10 "Footnote:     1 Includes the resident population for the 50 stat~        NA
```

There are a couple ways we could do this, but for now let's:

1. Make a new data frame with just P.R.
2. Remove the last 5 rows of the data frame.
3. Combine the two data frames.
4. Remove the P.R. dataframe from memory.

First, we use `filter()` to make a 1-row data frame with just Puerto Rico:

```r
puerto_rico_temp <- state_population |>
  filter(state_or_territory == "Puerto Rico")
puerto_rico_temp
```

```
# A tibble: 1 x 2
  state_or_territory population
  <chr>                   <dbl>
1 Puerto Rico           3285874
```

Second, we can use `head()` to select the first 51 rows of the data frame:

```r
state_population <- state_population |>
  head(51)
state_population
```

```
# A tibble: 51 x 2
   state_or_territory   population
   <chr>                     <dbl>
 1 Alabama                 5024279
 2 Alaska                   733391
 3 Arizona                 7151502
```

```
 4 Arkansas                  3011524
 5 California               39538223
 6 Colorado                  5773714
 7 Connecticut               3605944
 8 Delaware                   989948
 9 District of Columbia       689545
10 Florida                  21538187
# i 41 more rows
```

Third, we row-bind the two data frames together:

```
state_population <- state_population |>
  bind_rows(puerto_rico_temp)

state_population
```

```
# A tibble: 52 x 2
   state_or_territory   population
   <chr>                     <dbl>
 1 Alabama                 5024279
 2 Alaska                   733391
 3 Arizona                 7151502
 4 Arkansas                3011524
 5 California             39538223
 6 Colorado                5773714
 7 Connecticut             3605944
 8 Delaware                 989948
 9 District of Columbia     689545
10 Florida                21538187
# i 42 more rows
```

When we look at the tail of the data frame, we can see that Puerto Rico is now included.

## 2.13 Deleting data with `rm()`

Finally, we remove the temporary data frame from memory using `rm()`, which is short for "remove":

```
rm(puerto_rico_temp)
```

### 2.13.1 Check your knowledge

Review the functions we've learned so far. What do each of these do?

1. pivot_wider()
2. pivot_longer()
3. arrange()
4. group_by()
5. summarize()

### 2.13.2 Homework

1. Find a country's statistical office website
2. Find an interesting data set
3. Use download.file() to download the file
4. Load the file as a data frame in R
5. Clean the data as necessary, to show something interesting about this country.
6. Screenshot the final data table and email it to me, along with the code used to produce the document, titled `week_3_homework_(your_name).R`. We will present these in the next class.

# 3 Tidyverse 3: Data tips & tricks

## 3.1 Review: loading data, head(), tail()

This week is all about practice. I want to make sure you understand the basics before we start making charts, maps and websites.

Let's clean some data together.

1. Make a new file
2. Save your file
3. Load the Tidyverse
4. … and let's get started. Here's a link to a data set:

You can get the download link, usually, by finding the button, right clicking, and hitting "Copy link address". This will be slightly different depending on your browser.


Figure 3.1: You need to make sure you have a link to the file you want.

1. Download it into your project folder.

```
download.file("https://dam-api.bfs.admin.ch/hub/api/dam/assets/32007752/master", "input_da
```

Note: On Windows, you need to make an adjustment. You need to add `mode = "wb"` to the `download.file()` function.

```
download.file("https://dam-api.bfs.admin.ch/hub/api/dam/assets/32007752/master", "input_da
```

2. Load this data set into your R session

```
births <- read_csv("input_data/births.csv")
```

3. Take a look at the first 20 rows of the data set. Can you figure out what each column is?

```
births |> head()
```

```
# A tibble: 6 x 5
   YEAR CANTON AGE_MOTHER SEX_CHILD OBS_VALUE
  <dbl> <chr>  <chr>      <chr>         <dbl>
1  1969 CH     _T         T            102520
2  1969 CH     _T         F             49990
3  1969 CH     _T         M             52530
4  1969 CH     Y10T14     T                 6
5  1969 CH     Y10T14     F                 3
6  1969 CH     Y10T14     M                 3
```

4. Now look at the last 20 rows. What are some of the steps we're going to have to take?

```r
births |> tail()
```

```
# A tibble: 6 x 5
   YEAR CANTON AGE_MOTHER SEX_CHILD OBS_VALUE
  <dbl> <chr>  <chr>      <chr>         <dbl>
1  2023 26     Y60T64     T                 0
2  2023 26     Y60T64     F                 0
3  2023 26     Y60T64     M                 0
4  2023 26     Y65T69     T                 0
5  2023 26     Y65T69     F                 0
6  2023 26     Y65T69     M                 0
```

Can we guess what every column is?

## 3.2 Classwork: filter()

To review, the filter() function will sort out the data you want from the stuff you don't. For example, if you only wanted a dataset for boys in Zürich in 1970, you could use the following code:

```r
births |>
  filter(YEAR == 1970) |>
  filter(CANTON == 1) |>
  filter(SEX_CHILD == "M")
```

```
# A tibble: 13 x 5
    YEAR CANTON AGE_MOTHER SEX_CHILD OBS_VALUE
   <dbl> <chr>  <chr>      <chr>         <dbl>
 1  1970 1      _T         M              8330
 2  1970 1      Y10T14     M                 0
 3  1970 1      Y15T19     M               279
```

```
 4  1970 1      Y20T24     M              2274
 5  1970 1      Y25T29     M              3119
 6  1970 1      Y30T34     M              1745
 7  1970 1      Y35T39     M               768
 8  1970 1      Y40T44     M               133
 9  1970 1      Y45T49     M                12
10  1970 1      Y50T54     M                 0
11  1970 1      Y55T59     M                 0
12  1970 1      Y60T64     M                 0
13  1970 1      Y65T69     M                 0
```

Please answer the following questions: Your answer should be a code block that uses the **filter()** function to find the answer.

1. How many babies were born in Vaud (Canton number 22) in 2020?
2. How many girls were born in Zurich (Canton number 1) to mothers aged 30-34 in 2019?
3. Between 2000 and 2020, How many years did Ticino (Canton number 21) have fewer than 1400 boys born?
4. How many boys and girls were born in Switzerland in 2015?

## 3.3 Classwork: select(), rename(), mutate()

Let's clean up the data set.

1. TYPING IN ALL CAPS IS ANNOYING. Rename the columns to `year`, `canton`, `age_of_mother`, `sex_of_child`, `total_born`. You can use `rename()` or `colnames() <- c()`. It should look like this:

```
# A tibble: 10 x 5
    year canton age_of_mother sex_of_child total_born
   <dbl> <chr>  <chr>         <chr>             <dbl>
 1  1969 CH     _T            T                102520
 2  1969 CH     _T            F                 49990
 3  1969 CH     _T            M                 52530
 4  1969 CH     Y10T14        T                     6
 5  1969 CH     Y10T14        F                     3
 6  1969 CH     Y10T14        M                     3
 7  1969 CH     Y15T19        T                  3648
 8  1969 CH     Y15T19        F                  1780
 9  1969 CH     Y15T19        M                  1868
10  1969 CH     Y20T24        T                 30230
```

2. Let's say I don't care about gender. I want to know the total number of children born in each Canton each year to mothers of different ages. **Filter** only the total number of children born, and discard the boy and girl counts. It should look like this:

```
# A tibble: 10 x 5
    year canton age_of_mother sex_of_child total_born
   <dbl> <chr>  <chr>         <chr>             <dbl>
 1  1969 CH     _T            T                102520
 2  1969 CH     Y10T14        T                     6
 3  1969 CH     Y15T19        T                  3648
 4  1969 CH     Y20T24        T                 30230
 5  1969 CH     Y25T29        T                 36206
 6  1969 CH     Y30T34        T                 20479
 7  1969 CH     Y35T39        T                  9077
 8  1969 CH     Y40T44        T                  2633
 9  1969 CH     Y45T49        T                   240
10  1969 CH     Y50T54        T                     1
```

3. Now that `sex_of_child` column is pretty useless, isn't it? Let's **select()** only the columns that we care about. It should look like this:

```
# A tibble: 10 x 4
    year canton age_of_mother total_born
   <dbl> <chr>  <chr>              <dbl>
 1  1969 CH     _T                102520
 2  1969 CH     Y10T14                 6
 3  1969 CH     Y15T19              3648
 4  1969 CH     Y20T24             30230
 5  1969 CH     Y25T29             36206
 6  1969 CH     Y30T34             20479
 7  1969 CH     Y35T39              9077
 8  1969 CH     Y40T44              2633
 9  1969 CH     Y45T49               240
10  1969 CH     Y50T54                 1
```

4. The `age_of_mother` column is a bit of a mess. Let's clean it up. A function called `str_sub()` can help us with this. Learn how it works by typing `?str_sub` into the console. You can also just experiment with it with a test string.

```
str_sub("Maybe sub_str stands for submarine_string.", 1, 5)
str_sub("What about substitute_string?", 12, -2)
str_sub("Nah, maybe I'm overthinking it.", 16, 27)
```

```
[1] "Maybe"
```

```
[1] "substitute_string"
```

```
[1] "overthinking"
```

Now that you understand the function, you can use it to `mutate()` the `age_of_mother` column, and make two new columns, called `mother_age_from` and `mother_age_to`. It should look like this:

```
# A tibble: 19,305 x 6
    year canton age_of_mother total_born mother_age_from mother_age_to
   <dbl> <chr>  <chr>              <dbl> <chr>           <chr>
 1  1969 CH     _T                102520 T               ""
 2  1969 CH     Y10T14                 6 10              "14"
 3  1969 CH     Y15T19              3648 15              "19"
 4  1969 CH     Y20T24             30230 20              "24"
 5  1969 CH     Y25T29             36206 25              "29"
 6  1969 CH     Y30T34             20479 30              "34"
 7  1969 CH     Y35T39              9077 35              "39"
 8  1969 CH     Y40T44              2633 40              "44"
 9  1969 CH     Y45T49               240 45              "49"
10  1969 CH     Y50T54                 1 50              "54"
# i 19,295 more rows
```

5. It looks like we have more filtering to do. the `canton` column also includes the total number of births in Switzerland. We don't want that. Let's use **filter()** to remove the rows where the `canton` column is equal to "CH". We also want to get rid of the rows where the age_of_mother is "_T". It should look like this:

```
# A tibble: 17,160 x 6
    year canton age_of_mother total_born mother_age_from mother_age_to
   <dbl> <chr>  <chr>              <dbl> <chr>           <chr>
 1  1969 1      Y10T14                 0 10              14
 2  1969 1      Y15T19               532 15              19
 3  1969 1      Y20T24              4608 20              24
 4  1969 1      Y25T29              6149 25              29
 5  1969 1      Y30T34              3535 30              34
 6  1969 1      Y35T39              1423 35              39
 7  1969 1      Y40T44               366 40              44
 8  1969 1      Y45T49                25 45              49
 9  1969 1      Y50T54                 0 50              54
10  1969 1      Y55T59                 0 55              59
# i 17,150 more rows
```

6. But wait! Notice that the `mother_age_from` and `mother_age_to` columns are still characters. It's still just a string that looks like a number. We can convert them to integers with the `as.integer()` function. You should mutate over the column again, using the as.integer() function. It should look like this:

```
# A tibble: 17,160 x 6
    year canton age_of_mother total_born mother_age_from mother_age_to
   <dbl> <chr>  <chr>              <dbl>           <int>         <int>
 1  1969 1      Y10T14                 0              10            14
 2  1969 1      Y15T19               532              15            19
 3  1969 1      Y20T24              4608              20            24
 4  1969 1      Y25T29              6149              25            29
 5  1969 1      Y30T34              3535              30            34
 6  1969 1      Y35T39              1423              35            39
 7  1969 1      Y40T44               366              40            44
 8  1969 1      Y45T49                25              45            49
 9  1969 1      Y50T54                 0              50            54
10  1969 1      Y55T59                 0              55            59
# i 17,150 more rows
```

7. Now we don't need the `age_of_mother` column anymore. Let's **select()** only the columns that we care about. It should look like this:

```
# A tibble: 17,160 x 5
    year canton total_born mother_age_from mother_age_to
   <dbl> <chr>       <dbl>           <int>         <int>
 1  1969 1               0              10            14
 2  1969 1             532              15            19
 3  1969 1            4608              20            24
 4  1969 1            6149              25            29
 5  1969 1            3535              30            34
 6  1969 1            1423              35            39
 7  1969 1             366              40            44
 8  1969 1              25              45            49
 9  1969 1               0              50            54
10  1969 1               0              55            59
# i 17,150 more rows
```

8. Now that we have a clean data set, let's save it to a variable, if you haven't been doing that already. This time, I think it's safe to overwrite the original value, so we simply use `->` to give it the same name as before. It should look like this:

**Before**

```
births |>
    rename(
    year = YEAR,
    ...
    ...
    ...
```

**After**

```
births <- births |>
    rename(
    year = YEAR,
    ...
    ...
    ...
```

## 3.4 Joining two datasets together with `left_join()`

There's one last annoying thing about this data set: the canton numbers. It would be really annoying to have to remember that Zurich is canton number 1, and so on. We can fix this by joining the data set with another data set that has the canton names.

We can find the canton names and numbers here:

https://www.bfs.admin.ch/asset/de/453856

Can you find the download link for the canton names and numbers? Use `download.file()` to download it into your project folder.

Now import the data set into your R session. This one is especially messy, so I wrote some code to help you out. You can just copy and paste this code into your own file if you like.

```
library(readxl)

canton_names <- read_excel("input_data/canton_nums.xlsx")

canton_names <- canton_names |>
  select(1:2) |> # I select the first two columns because the rest are filled with junk.
  tail(-4) |> # I delete the first four rows, because the first four are metadata.
  head(26) # I select only the first 26 rows, because the rest are metadata.

# Now I rename the columns to something more useful.
colnames(canton_names) <- c("bfs_canton_number", "canton_name")

canton_names
```

```
# A tibble: 26 x 2
   bfs_canton_number canton_name
   <chr>             <chr>
 1 1                 Zürich
 2 2                 Bern
 3 3                 Luzern
 4 4                 Uri
 5 5                 Schwyz
 6 6                 Obwalden
 7 7                 Nidwalden
 8 8                 Glarus
 9 9                 Zug
10 10                Freiburg
# i 16 more rows
```

Now the tricky part: joining. there's several different functions to join things, but the one we'll use is `left_join()`. This function takes two data sets, and joins them together into one. It's called "left join" because the data set on the left side of the function is the one that will be kept, and the data set on the right side will be joined to it.

Now we use `left_join()` to join the two data sets together. This function takes two arguments: the first is the data set you want to join, the second is `by=`, in which you put the names of the columns you want to join by. In this case, we want to join by the `canton_number` column in the `births` data set, and the `bfs_canton_number` column in the `canton_names` data set.

When we do this, it will match all the rows in the `births` data set with the corresponding row in the `canton_names` data set. If there's no match, it will put `NA` in the column, meaning that there is no data there.

```
births |>
  left_join(canton_names, by = c("canton" = "bfs_canton_number"))
```

```
# A tibble: 17,160 x 6
    year canton total_born mother_age_from mother_age_to canton_name
   <dbl> <chr>       <dbl>           <int>         <int> <chr>
 1  1969 1               0              10            14 Zürich
 2  1969 1             532              15            19 Zürich
 3  1969 1            4608              20            24 Zürich
 4  1969 1            6149              25            29 Zürich
 5  1969 1            3535              30            34 Zürich
 6  1969 1            1423              35            39 Zürich
 7  1969 1             366              40            44 Zürich
 8  1969 1              25              45            49 Zürich
 9  1969 1               0              50            54 Zürich
10  1969 1               0              55            59 Zürich
```

Table 3.1: Two tables

```
# A tibble: 7 x 2
  name    student_number
  <chr>   <chr>
1 Urs     101
2 Rebekka 102
3 Dario   103
4 Jörg    104
5 Maude   105
6 Daniel  206
7 Mark    207
```

```
# A tibble: 10 x 2
   student_number grade
            <dbl> <dbl>
 1            101    95
 2            102    85
 3            103    90
 4            104   100
 5            105    90
 6            106    90
 7            107    85
 8            108    70
 9            109    60
10            110    55
```

```
# i 17,150 more rows
```

When this looks good, save it to a variable.

```
births <- births |>
  left_join(canton_names, by = c("canton" = "bfs_canton_number"))
```

Make sure that the data types are the same! If they're not, you need to use a function to convert them using `mutate()`. Some functions that can do this are:

1. `as.integer()` 0, 1, 2, 3
2. `as.numeric()` 0.0, 1.0, 2.0, 3.0
3. `as.logical()` FALSE, TRUE, TRUE, TRUE
4. `as.character()` "0", "one", "2", "Zürich"
5. `as.roman()` I, II, III, IV

For example, here are two data sets that we want to join together. The first data set is a list of students, and the second data set is a list of grades. We want to join them together by the student number, to see which students got which grades. But it won't work!

```
# table on the left
students
# table on the right
grades
```

In the first table, the `student_number` column is a character, and in the second table, the `student_number` column is a number. In RStudio, you can see the data types when you print to the console. We need to convert one of them so that they match.

```r
students |>
  mutate(student_number = as.numeric(student_number))
```

```
# A tibble: 7 x 2
  name    student_number
  <chr>            <dbl>
1 Urs                101
2 Rebekka            102
3 Dario              103
4 Jörg               104
5 Maude              105
6 Daniel             206
7 Mark               207
```

Now, we can join them together. Here, the column names are the same, so we can simplify the `by=` argument.

```r
students |>
  mutate(student_number = as.numeric(student_number)) |>
  left_join(grades, by = "student_number")
```

```
# A tibble: 7 x 3
  name    student_number grade
  <chr>            <dbl> <dbl>
1 Urs                101    95
2 Rebekka            102    85
3 Dario              103    90
4 Jörg               104   100
5 Maude              105    90
6 Daniel             206    NA
7 Mark               207    NA
```

## 3.5 Other types of join: `inner_join()`, `right_join()` and `full_join()`

In this example, you'll notice that there are some students who don't have grades. This is because they're not in the `grades` data set. Additionally, there were some grades that weren't associated with a student, because they're not in the `students` data set.

`left_join()` is called a "left join" because the data on the left (the first argument) will be kept, but the stuff on the right will be dropped if there isn't a match. However, there are some other kinds of joins that you can use.

1. `inner_join()` will only keep the rows that have a match in both data sets. If there's no match, it will be dropped.

```
students |>
  mutate(student_number = as.numeric(student_number)) |>
  inner_join(grades, by = "student_number")
```

```
# A tibble: 5 x 3
  name    student_number grade
  <chr>            <dbl> <dbl>
1 Urs                101    95
2 Rebekka            102    85
3 Dario              103    90
4 Jörg               104   100
5 Maude              105    90
```

2. `right_join()` is the opposite of `left_join()`. It will keep the data on the right, and drop the data on the left if there isn't a match.

```
students |>
  mutate(student_number = as.numeric(student_number)) |>
  right_join(grades, by = "student_number")
```

```
# A tibble: 10 x 3
   name    student_number grade
   <chr>            <dbl> <dbl>
 1 Urs                101    95
 2 Rebekka            102    85
 3 Dario              103    90
 4 Jörg               104   100
 5 Maude              105    90
 6 <NA>               106    90
 7 <NA>               107    85
 8 <NA>               108    70
 9 <NA>               109    60
10 <NA>               110    55
```

3. `full_join()` will keep all the data, even if there isn't a match. If there isn't a match, it will put `NA` in the column.

```
students |>
  mutate(student_number = as.numeric(student_number)) |>
  full_join(grades, by = "student_number")
```

```
# A tibble: 12 x 3
   name    student_number grade
   <chr>            <dbl> <dbl>
 1 Urs                101    95
 2 Rebekka            102    85
 3 Dario              103    90
 4 Jörg               104   100
 5 Maude              105    90
 6 Daniel             206    NA
 7 Mark               207    NA
 8 <NA>               106    90
 9 <NA>               107    85
10 <NA>               108    70
11 <NA>               109    60
12 <NA>               110    55
```

I find myself using `left_join()` probably 95% of the time, but it's good to know that there are other options.

## 3.6 Dealing with missing data with `replace_na()` and `drop_na()`

The new data we've created has some missing data. For example, some students don't have grades, and some grades don't have students. Missing data in R is represented by `NA`, and can create some problems for you. There are two ways to deal with this: you can either **replace** the missing data with something else, or you can **drop** the rows with missing data.

To replace missing data, you can use the `replace_na()` function. This function takes two arguments: the first is the data set you want to replace the missing data in, and the second is the value you want to replace the missing data with. For example, if you want to replace all the missing data in the `name` column with "NO NAME FOUND", you can use the following code:

```
students |>
  mutate(student_number = as.numeric(student_number)) |>
  full_join(grades, by = "student_number") |>
  mutate(name = replace_na(name, "NO NAME FOUND"))
```

```
# A tibble: 12 x 3
   name          student_number grade
   <chr>                  <dbl> <dbl>
 1 Urs                      101    95
 2 Rebekka                  102    85
 3 Dario                    103    90
```

```
 4 Jörg                         104   100
 5 Maude                        105    90
 6 Daniel                       206    NA
 7 Mark                         207    NA
 8 NO NAME FOUND                106    90
 9 NO NAME FOUND                107    85
10 NO NAME FOUND                108    70
11 NO NAME FOUND                109    60
12 NO NAME FOUND                110    55
```

Second, you can use the `drop_na()` function to drop the rows with missing data. This function takes one argument: the data set you want to drop the missing data from. For example, if you want to drop all the rows with missing data in the `grade` column, you can use the following code:

```
students |>
  mutate(student_number = as.numeric(student_number)) |>
  full_join(grades, by = "student_number") |>
  mutate(name = replace_na(name, "NO NAME FOUND")) |>
  drop_na(grade)
```

```
# A tibble: 10 x 3
   name          student_number grade
   <chr>                  <dbl> <dbl>
 1 Urs                      101    95
 2 Rebekka                  102    85
 3 Dario                    103    90
 4 Jörg                     104   100
 5 Maude                    105    90
 6 NO NAME FOUND            106    90
 7 NO NAME FOUND            107    85
 8 NO NAME FOUND            108    70
 9 NO NAME FOUND            109    60
10 NO NAME FOUND            110    55
```

## 3.7 Review together: group_by(), summarize()

Let's go back to our cleaned birth data set and do some analysis.

For example, suppose we want to know how many children were born each year to women over 45 years old. First, we need to **filter()** only the rows where the `mother_age_to` column is greater than 45.

```r
births |>
  filter(mother_age_to > 45)
```

```
# A tibble: 7,150 x 6
    year canton total_born mother_age_from mother_age_to canton_name
   <dbl> <chr>       <dbl>           <int>         <int> <chr>
 1  1969 1              25              45            49 Zürich
 2  1969 1               0              50            54 Zürich
 3  1969 1               0              55            59 Zürich
 4  1969 1               0              60            64 Zürich
 5  1969 1               0              65            69 Zürich
 6  1969 2              26              45            49 Bern
 7  1969 2               0              50            54 Bern
 8  1969 2               0              55            59 Bern
 9  1969 2               0              60            64 Bern
10  1969 2               0              65            69 Bern
# i 7,140 more rows
```

Now let's think. We want to know how many children were born each year. We need to **group_by()** the `year` column.

```r
births |>
  filter(mother_age_to > 45) |>
  group_by(year)
```

```
# A tibble: 7,150 x 6
# Groups:   year [55]
    year canton total_born mother_age_from mother_age_to canton_name
   <dbl> <chr>       <dbl>           <int>         <int> <chr>
 1  1969 1              25              45            49 Zürich
 2  1969 1               0              50            54 Zürich
 3  1969 1               0              55            59 Zürich
 4  1969 1               0              60            64 Zürich
 5  1969 1               0              65            69 Zürich
 6  1969 2              26              45            49 Bern
 7  1969 2               0              50            54 Bern
 8  1969 2               0              55            59 Bern
 9  1969 2               0              60            64 Bern
10  1969 2               0              65            69 Bern
# i 7,140 more rows
```

Now, we want to **summarize()** the data. We want to know the total number of children born each year. We can use the `sum()` function to do this.

```r
births |>
  filter(mother_age_to > 45) |>
  group_by(year) |>
  summarize(total_born = sum(total_born))
```

```
# A tibble: 55 x 2
    year total_born
   <dbl>      <dbl>
 1  1969        241
 2  1970        226
 3  1971        196
 4  1972        180
 5  1973        153
 6  1974        131
 7  1975        102
 8  1976         96
 9  1977         83
10  1978         79
# i 45 more rows
```

## 3.8 Homework: Answering questions with group_by() and summarize()

Finally, use this data to answer the following questions:

1. How many teenage births were there in Zürich between 2000 and 2020?
2. What is the approximate age of oldest woman to ever give birth in each canton?
3. How many children were born in Zurich, Bern and Geneva in 2019?
4. How many children were born in each Canton in 1980?
5. In each canton, what was the most common age range for mothers to give birth in 1970, 1990, and 2010?

Please email me the code you used to find the answers in a document named `week_3_homework_(your_name).R` by Tuesday, March 12th.

## 3.9 Bonus questions

For people with some experience working with data, these are a bit easy. If you'd like more practice, here are some bonus questions:

These aren't part of the homework, but might be a good challenge for you.

6. Between 2010 and 2020, what were the average number of children born each year in each canton?
7. Building off this, which years had an above average birth rate for that decade?
8. Here's some data about the number of deaths in each canton. https://opendata.swiss/de/dataset/todesfalle-nach-funf-jahres-altersgruppe-geschlecht-und-kanton-1969-2023 Can you download and clean this one?
9. Simplify the births data into just number of births by year. Then **join()** this data with the deaths data. What was the total population change in Zürich in 2000? (Excluding immigration and emmigration)
10. Make a plot of the total births and deaths in Basel-Stadt between 1970 and 2000.

# 4 Finding your own data sets

```r
library(tidyverse)
```

So far, we've been relying on some pretty simple example data, mainly from the Swiss government. This week, we're going to look at how you can find your own data sets, so you can work on projects that are interesting to you.

### 4.0.1 Using national data portals

Just about every country has some sort of national data portal or statistical bureau that collects and publishes data. These are great places to find data, because the data is usually well-organized and reliable. A couple examples are:

- Canada: https://open.canada.ca/en
- Germany: https://www-genesis.destatis.de/genesis/online
- Taiwan: https://data.gov.tw/en

## 4.1 Classwork: Finding demographic data

1. Think of a place that you're interested in (besides Switzerland, because we've done a lot of this already)

2. Find the website of the statistical bureau or data portal of that country.

3. Find a dataset that contains the following information:

   - Population of each state / province / canton / region
   - Something related to the economy (GDP, unemployment, etc.)
   - Something related to health (life expectancy, infant mortality, etc.)
   - Something related to education (literacy rate, school enrollment, etc.)
   - Something related to the environment (CO2 emissions, forest cover, etc.)

4. Download the data and load it into R.

### 4.1.1 Effectively searching for data

When you're looking for data, it's important to use the right search terms. Here are a few tips:

- Use the `filetype:` operator to search for specific file types. For example, if you're looking for CSV files using Google, you can search for **filetype:csv canadian housing**, and it will only return CSV files.
- Use the `site:` operator to search within a specific website. For example, if you're looking for data on the Swiss government's website, you can search for **site:admin.ch population**.
- Use the `intitle:` operator to search for specific words in the title of a page. For example, if you're looking for data on the Swiss government's website, you can search for **intitle:migration site:admin.ch**.

## 4.2 R Packages that supply data

There are a few R packages that can help you get data from various sources. Here are a few of the more useful ones:

### 4.2.1 Eurostat

Eurostat is the statistical office of the European Union. They collect data on a wide range of topics, including agriculture, trade, and the environment. You can access their data using the `eurostat` package, which you might have to install seperately.

```r
install.packages("eurostat")
```

```r
library(eurostat)
```

This package has a `search_eurostat` function that you can use to search for data sets. For example, if you're interested in animal statistics, you can search for "Animal". This will return a data frame with the results of the search, including codes for the data sets.

```r
animal_stats <- search_eurostat("Animal")
animal_stats
```

```
# A tibble: 6 x 9
  title        code  type  last.update.of.data last.table.structure~1 data.start
  <chr>        <chr> <chr> <chr>               <chr>                  <chr>
1 Animal popu~ agr_~ data~ 26.02.2025          02.12.2024             1977
```

```
2 Animal hous~ ef_a~ data~ 28.08.2024          28.08.2024               2010
3 Animal hous~ ef_a~ data~ 16.07.2024          16.07.2024               2010
4 Animal hous~ ef_a~ data~ 16.07.2024          16.07.2024               2010
5 Animal popu~ agr_~ data~ 26.02.2025          02.12.2024               1977
6 Animal popu~ agr_~ data~ 26.02.2025          02.12.2024               1977
# i abbreviated name: 1: last.table.structure.change
# i 3 more variables: data.end <chr>, values <dbl>, hierarchy <dbl>
```

However, I've always found this to be a bit janky and not very useful. You might have better luck searching the website.

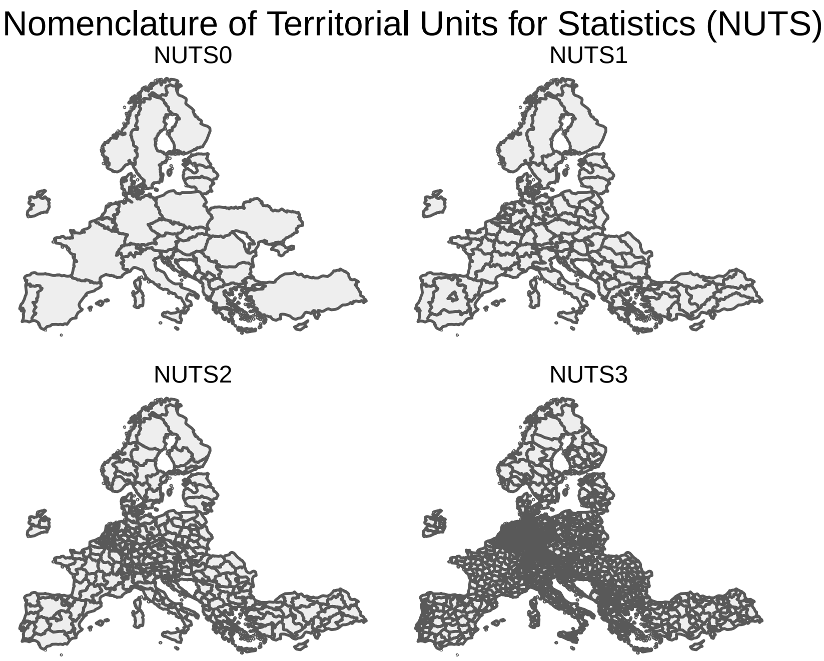
https://ec.europa.eu/eurostat

In either case, you just want to find the code for the data set you're interested in. Once you have that, you can use the `get_eurostat` function to download the data.

```r
animal_data <- get_eurostat("agr_r_animal")
animal_data
```

```
# A tibble: 30 x 7
    ...1 freq  animals unit   geo   TIME_PERIOD values
   <dbl> <chr> <chr>   <chr>  <chr> <date>       <dbl>
 1     1 A     A2000   THS_HD AT    1977-01-01   2547.
 2     2 A     A2000   THS_HD AT    1978-01-01   2594.
 3     3 A     A2000   THS_HD AT    1979-01-01   2548.
 4     4 A     A2000   THS_HD AT    1980-01-01   2517.
 5     5 A     A2000   THS_HD AT    1981-01-01   2530.
 6     6 A     A2000   THS_HD AT    1982-01-01   2546.
 7     7 A     A2000   THS_HD AT    1983-01-01   2633.
 8     8 A     A2000   THS_HD AT    1984-01-01   2669.
 9     9 A     A2000   THS_HD AT    1985-01-01   2651.
10    10 A     A2000   THS_HD AT    1986-01-01   2637.
# i 20 more rows
```

Eurostat often uses a geographical division called the NUTS (Nomenclature of Territorial Units for Statistics) system. This system divides countries into regions, which are then divided into smaller regions, and so on. The idea is to have a consistent way of dividing up countries into similar-sized areas for statistical purposes. Often, this doesn't correspond to any administrative divisions, but it's useful for comparing regions across countries.




### 4.2.2 World Bank Statistics

Next, we have the `wbstats` package, which allows you to access data from the World Bank. This can give you a lot of economic data between countries.

```
install.packages("wbstats")
```

```
library(wbstats)
```

Like Eurostat, the `wb_search` function allows you to search for data sets

```
wb_search("electricity")
```

```
# A tibble: 127 x 3
   indicator_id                     indicator                        indicator_desc
   <chr>                            <chr>                            <chr>
 1 1.1_ACCESS.ELECTRICITY.TOT       Access to electricity (% of to~ Access to ele~
 2 1.2_ACCESS.ELECTRICITY.RURAL     Access to electricity (% of ru~ Access to ele~
 3 1.3_ACCESS.ELECTRICITY.URBAN     Access to electricity (% of ur~ Access to ele~
 4 2.0.cov.Ele                      Coverage: Electricity            The coverage ~
 5 2.0.hoi.Ele                      HOI: Electricity                 The Human Opp~
 6 4.1.1_TOTAL.ELECTRICITY.OUTPUT Total electricity output (GWh)  Total electri~
 7 4.1.2_REN.ELECTRICITY.OUTPUT     Renewable energy electricity o~ Renewable ene~
 8 4.1_SHARE.RE.IN.ELECTRICITY      Renewable electricity (% in to~ Renewable ele~
```

```
 9 9060000                          9060000:ACTUAL HOUSING, WATER,~ <NA>
10 BM.GSR.TRAN.ZS                   Transport services (% of servi~ Transport cov~
# i 117 more rows
```

And like Eurostat, you can use the `wb_data` function to download the data.

```r
wb_data("4.1.1_TOTAL.ELECTRICITY.OUTPUT")
```

```
# A tibble: 30 x 10
    ...1 iso2c iso3c country  date 4.1.1_TOTAL.ELECTRICITY.OU~1 unit  obs_status
   <dbl> <chr> <chr> <chr>   <dbl>                        <dbl> <lgl> <lgl>
 1     1 AW    ABW   Aruba    1990                          338 NA    NA
 2     2 AW    ABW   Aruba    1991                          339 NA    NA
 3     3 AW    ABW   Aruba    1992                          341 NA    NA
 4     4 AW    ABW   Aruba    1993                          531 NA    NA
 5     5 AW    ABW   Aruba    1994                          564 NA    NA
 6     6 AW    ABW   Aruba    1995                          616 NA    NA
 7     7 AW    ABW   Aruba    1996                          642 NA    NA
 8     8 AW    ABW   Aruba    1997                          675 NA    NA
 9     9 AW    ABW   Aruba    1998                          730 NA    NA
10    10 AW    ABW   Aruba    1999                          738. NA    NA
# i 20 more rows
# i abbreviated name: 1: `4.1.1_TOTAL.ELECTRICITY.OUTPUT`
# i 2 more variables: footnote <lgl>, last_updated <date>
```

### 4.2.3 BFS data

If you're going to be working frequently with Swiss data, you can use an R package built by the Swiss Federal Statistical Office (BFS) to access their data.

```r
install.packages("BFS")
```

```r
library(BFS)
```

This works essentially the same as the last two; you can search for data sets using the `bfs_get_catalog_data` function, and download data using the `bfs_get_data` function.

```r
bfs_get_catalog_data(language = "en", extended_search = "university")
```

```
# A tibble: 6 x 6
  title                      number_bfs language number_asset publication_date url_px
  <chr>                      <chr>      <chr>    <chr>        <date>           <chr>
```

```
1 Businesses by diffic~ px-x-0602~ en       33947196     2025-02-24       https~
2 Businesses by diffic~ px-x-0602~ en       33947195     2025-02-24       https~
3 University of applie~ px-x-1502~ en       31306033     2024-03-28       https~
4 University of applie~ px-x-1502~ en       31306029     2024-03-28       https~
5 University students ~ px-x-1502~ en       31305852     2024-03-28       https~
6 University students ~ px-x-1502~ en       31305854     2024-03-28       https~
```

```
bfs_get_data(number_bfs = "px-x-1502040100_132", language = "en") |> write_csv("input_data
```

```
# A tibble: 30 x 5
   Year    `ISCED Field`     `Citizenship (category)` `Level of study`
   <chr>   <chr>             <chr>                    <chr>
 1 1990/91 Education science Switzerland              First university degree o~
 2 1990/91 Education science Switzerland              Bachelor
 3 1990/91 Education science Switzerland              Master
 4 1990/91 Education science Switzerland              Doctorate
 5 1990/91 Education science Switzerland              Further education, advanc~
 6 1990/91 Education science Foreign country          First university degree o~
 7 1990/91 Education science Foreign country          Bachelor
 8 1990/91 Education science Foreign country          Master
 9 1990/91 Education science Foreign country          Doctorate
10 1990/91 Education science Foreign country          Further education, advanc~
# i 20 more rows
# i 1 more variable: `University students` <dbl>
```

## 4.3 Web scraping

Our final method of acquiring data is the real wild west: web scraping.

Extracting usable data from websites, is a really, really big topic, and one that we can't really cover in depth in one lesson. However, we can do some basic web scraping that will get you pretty far. In this little bottled example, we'll scrape a table from Wikipedia, in this case a list of US cities by area.

To do this, we'll use the `rvest` package, which you might have to install.

```
install.packages("rvest")
```

```
library(rvest)
```

```
Attaching package: 'rvest'
```

```
The following object is masked from 'package:readr':

    guess_encoding
```

This gives us quite a few functions to download and parse HTML. We'll start by downloading the HTML of the page using the `read_html` function.

```r
html <- read_html("https://en.wikipedia.org/wiki/List_of_United_States_cities_by_area")

html
```

```
{html_document}
<html class="client-nojs vector-feature-language-in-header-enabled vector-feature-language
[1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
[2] <body class="skin--responsive skin-vector skin-vector-search-vue mediawik ...
```

This gives us the HTML of the page, just the code that makes up the website.

HTML is a markup language, which means it's a way of describing the structure of a document. It's made up of tags, which are enclosed in angle brackets. For example, <h1> is a tag that indicates a heading, <h2> is a subheading, and so on.

Here's a simple example:

```html
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <!-- The h1 tag is a heading-1, which is usually the title of the page -->
    <h1>My website about frogs</h1>

    <!-- The h2 tag is a heading-2, which is a subheading, usually denoting a section -->
    <h2>What is a frog?</h2>

    <!-- The p tag is a paragraph, which is used for text -->
    <p>A frog is a small amphibian that lives in water and on land.</p>

    <!-- You can have multiple tags of any type -->
    <h2>Where do frogs live?</h2>
    <p>Frogs live in ponds, rivers, and lakes.</p>

    <h2>What do frogs eat?</h2>
    <p>Frogs eat insects and other small animals.</p>
  </body>
</html>
```

When opened in a web browser, this would look like this:

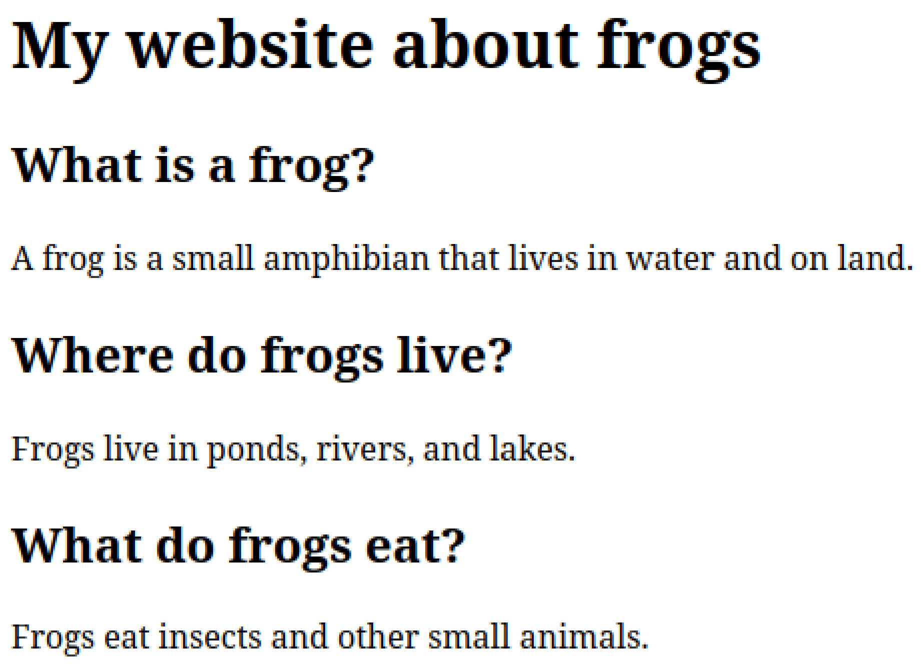



Figure 4.1: We are now web designers.

We can now use the `html_nodes` function to extract specific parts of the page. For example, to extract all the H1 tags, we can use the following code:

```
html |>
  html_nodes("h1")
```

```
{xml_nodeset (1)}
[1] <h1 id="firstHeading" class="firstHeading mw-first-heading"><span class=" ...
```

For H2 tags, we can use this code. To get all the text inside the tags, we can use the `html_text` function.

Note that there are multiple H2 tags on the page, so we get a list of them.

```
html |>
  html_nodes("h2") |>
  html_text()
```

```
[1] "Contents"           "List"               "Gallery"
[4] "See also"           "Explanatory notes"  "References"
```

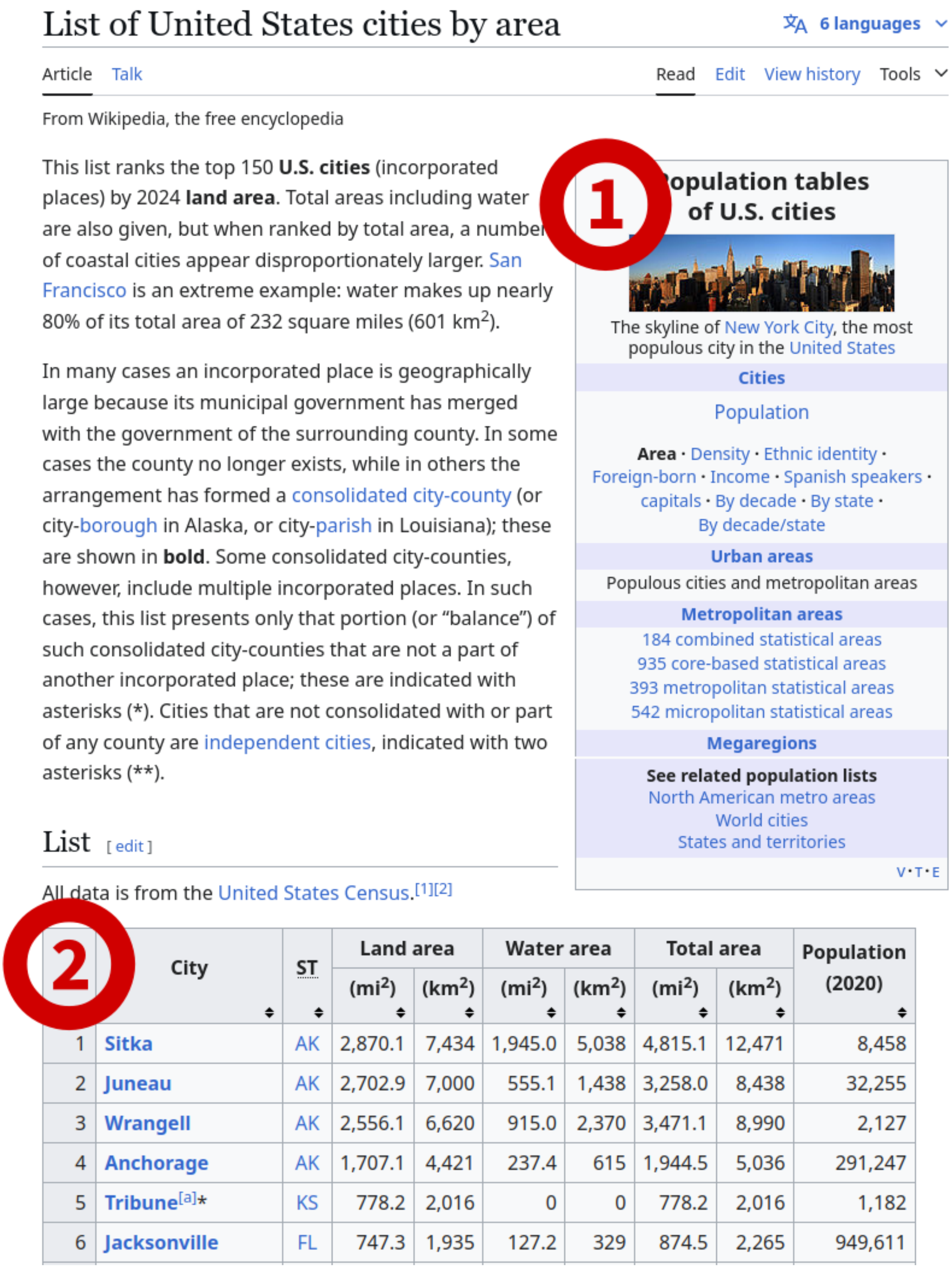
We can also extract tables from the page. To do this, we can use the `html_nodes` function with the `table` tag.

```
tables_on_page <- html |>
  html_nodes("table")

tables_on_page
```

```
{xml_nodeset (2)}
[1] <table class="sidebar nomobile nowraplinks"><tbody>\n<tr><th class="sideb ...
[2] <table class="sortable wikitable sticky-header-multi static-row-numbers s ...
```

Because there are multiple tables on the page, we get a list of them. We can extract the second table, which is the one we're interested in. You can go back to the website and count down to whatever table you want to extract, or just do it with trial-and-error.



# List of United States cities by area

6 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

This list ranks the top 150 **U.S. cities** (incorporated places) by 2024 **land area**. Total areas including water are also given, but when ranked by total area, a number of coastal cities appear disproportionately larger. San Francisco is an extreme example: water makes up nearly 80% of its total area of 232 square miles (601 km$^2$).

In many cases an incorporated place is geographically large because its municipal government has merged with the government of the surrounding county. In some cases the county no longer exists, while in others the arrangement has formed a consolidated city-county (or city-borough in Alaska, or city-parish in Louisiana); these are shown in **bold**. Some consolidated city-counties, however, include multiple incorporated places. In such cases, this list presents only that portion (or "balance") of such consolidated city-counties that are not a part of another incorporated place; these are indicated with asterisks (*). Cities that are not consolidated with or part of any county are independent cities, indicated with two asterisks (**).

**Population tables of U.S. cities**

The skyline of New York City, the most populous city in the United States

**Cities**

Population

**Area** · Density · Ethnic identity · Foreign-born · Income · Spanish speakers · capitals · By decade · By state · By decade/state

**Urban areas**

Populous cities and metropolitan areas

**Metropolitan areas**

184 combined statistical areas
935 core-based statistical areas
393 metropolitan statistical areas
542 micropolitan statistical areas

**Megaregions**

**See related population lists**
North American metro areas
World cities
States and territories

V · T · E

## List [ edit ]

All data is from the United States Census.[1][2]

| | City | ST | Land area | | Water area | | Total area | | Population (2020) |
|---|---|---|---|---|---|---|---|---|---|
| | | | (mi$^2$) | (km$^2$) | (mi$^2$) | (km$^2$) | (mi$^2$) | (km$^2$) | |
| 1 | **Sitka** | AK | 2,870.1 | 7,434 | 1,945.0 | 5,038 | 4,815.1 | 12,471 | 8,458 |
| 2 | **Juneau** | AK | 2,702.9 | 7,000 | 555.1 | 1,438 | 3,258.0 | 8,438 | 32,255 |
| 3 | **Wrangell** | AK | 2,556.1 | 6,620 | 915.0 | 2,370 | 3,471.1 | 8,990 | 2,127 |
| 4 | **Anchorage** | AK | 1,707.1 | 4,421 | 237.4 | 615 | 1,944.5 | 5,036 | 291,247 |
| 5 | **Tribune**[a]* | KS | 778.2 | 2,016 | 0 | 0 | 778.2 | 2,016 | 1,182 |
| 6 | **Jacksonville** | FL | 747.3 | 1,935 | 127.2 | 329 | 874.5 | 2,265 | 949,611 |

Figure 4.2: The table we want is the second one on the page.

The structure of this data is a little odd, because it's a list of lists. We extract the second element of the list, which is the table we want.

```
cities_table <- tables_on_page[[2]]
cities_table
```

```
{html_node}
<table class="sortable wikitable sticky-header-multi static-row-numbers sort-under col1lef
[1] <tbody>\n<tr>\n<th rowspan="2">City\n</th>\n<th rowspan="2">\n<abbr title ...
```

Finally, we can use the `html_table` function to convert this table into a data frame.

```
cities_df <- cities_table |>
  html_table()

cities_df
```

```
# A tibble: 151 x 9
   City     ST    `Land area` `Land area` `Water area` `Water area` `Total area`
   <chr>    <chr> <chr>       <chr>       <chr>        <chr>        <chr>
 1 City     ST    (mi2)       (km2)       (mi2)        (km2)        (mi2)
 2 Sitka    AK    2,870.1     7,434       1,945.0      5,038        4,815.1
 3 Juneau   AK    2,702.9     7,000       555.1        1,438        3,258.0
 4 Wrangell AK    2,556.1     6,620       915.0        2,370        3,471.1
 5 Anchora~ AK    1,707.1     4,421       237.4        615          1,944.5
 6 Tribune~ KS    778.2       2,016       0            0            778.2
 7 Jackson~ FL    747.3       1,935       127.2        329          874.5
 8 Anacond~ MT    736.7       1,908       4.7          12           741.4
 9 Butte *  MT    715.8       1,854       0.6          1.6          716.3
10 Houston  TX    640.8       1,660       31.2         81           672.0
# i 141 more rows
# i 2 more variables: `Total area` <chr>, `Population(2020)` <chr>
```

That's it! You've now scraped a table from Wikipedia. This is a very basic example, but you could use this basic concept to scrape data from any website that has tables on it.

### 4.3.1 Classwork: Scraping a table from the BBC

Here's a link to the BBC's election results page for the 2024 UK general election:

https://www.bbc.com/news/election/2024/uk/results

This page has a table with the results of the election. Your task is to scrape this table and load it into R as a data frame.

Your result should look something like this:

```
# A tibble: 34 x 3
   Party                     `Vote share` `Change since 2019`
   <chr>                     <chr>        <chr>
 1 Labour                    33.7%        +1.6%
 2 Conservative              23.7%        -19.9%
 3 Reform UK                 14.3%        +12.3%
 4 Liberal Democrat          12.2%        +0.7%
 5 Green                     6.7%         +4.0%
 6 Scottish National Party   2.5%         -1.4%
 7 Plaid Cymru               0.7%         +0.2%
 8 Sinn Fein                 0.7%         +0.1%
 9 Workers Party of Britain  0.7%         +0.7%
10 Democratic Unionist Party 0.6%         -0.2%
# i 24 more rows
```

### 4.3.2 Some web scraping tips

- **Be polite**: Don't scrape websites too often, and don't scrape them too fast. This can overload the server and get you banned.
- **Save the data**: Once you've scraped the data, save it to a file. This way, you don't have to scrape the website again. Websites can change, and you might not get the same data if you scrape it again.
- **Check the terms of service**: Some websites don't allow scraping. If you're doing something that isn't allowed, be extra careful. If you're selling the data, it could get you in some real legal trouble.

### 4.3.3 Next steps with web scraping

If you'd like to keep using R, the book "R for Data Science" has a great chapter on web scraping. You can find it here: https://r4ds.hadley.nz/webscraping.html

However, R is pretty limited when it comes to web scraping. If you're interested in doing more complicated things, I'd recommend learning Python or Go.