LiDaViR

An introduction to R. Maybe especially useful for Excel users

Leonardo Hansa

2021-08-13

Contents

Chapter 1

First steps

R is an environment for statistical analysis, including gathering, modelling and visualizing data. We also call R the programming language used on this environment (http://www.r-project.org/).

1.1 Programming in R

Why you should use R?

- Free and open
- It includes lots of tools for plots
- There is an interface, RStudio, which makes R projects very comfortable
- Available on Windows, Mac and Linux
- Console, scripts and notebooks available
- Updated with state-of-art methodologies
- Users community.

What trouble you will find.

- It is really difficult become an expert on every aspects of R
- Maintenance of codes will be hard if the language is used unproperly
- Execution can be very slow if the language is used unproperly
- Too much RAM can be needed

There are several tools that can be used for programming in R. The most spread one is RStudio. The company behind this software provides users with a very complete free version. They also work on several tools for R but independently from the R project.

Copy and pasting is a great way for developing good scripts. RStudio have written several cheatsheets with key content to getting started with some of the R tools. We may get some help from them. Go and visit their website.

1.2 R as a calculator

Our first approach to R will be using it as a calculator. Later on, we will begin taking advantage of it as a programming language. The main idea of these first lines is that we can compute both easy and complex calculations using the console of RStudio (some comments about RStudio IDE will be given during the lessons).

We can compute elementary mathematical operations, with common notation.

```
1 + 1
100 * 100
20 / 4
20 / 3
2 ** 10
2 ^ 10
```

For more complex operations we need *functions.* Functions (in any programming language) are pieces of code that make some operations. We'll get into them later on in the course. For now, it is enough knowing that R provide us with a lot of functions to make lots of calculations.

```
## [1] 120
```

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

Suppose we want to calculate the force that is needed if we want to move an object of 9.5 kg with an accelaration of 4.7 m/s^2. Newton's second law may help: $F = m \cdot a$.

```
9.5 * 4.7
```

```
## [1] 44.65
```

Yeah! Cool, we did it! We now know that 44.65 N are needed to move the object. The thing is that we will need to repeat the operation every time we desire to see the result. Avoiding that repetition can be achieved giving a name to that operation. `newton` seems a nice selection. We do this by using the `<-` symbol.

```
newton <- 9.5 * 4.7
newton
```

```
## [1] 44.65
```

Now, every time we write `newton` on the console we will get that result. In fact, on the upper right side of our RStudio windows we will see that something called `newton` has appeared, with a value of 44.65.

This idea can also be applied to the mass and the acceleration. We should define *something* with the mass value and another *something* with the acceleration value. In the future we will understand why this is so important.

```
acc <- 4.7
mass <- 9.5
newton <- mass * acc
newton
```

```
## [1] 44.65
```

Exercises.

- Change the values of `acc` and `mass` (make sure on the upper right corner of RStudio that you have actually changed them). Write on the console `newton`. Try to understand what is necessary for successfully getting that changed.
- What happens if you use `=` instead of `<-`? We will get into this by the end of the course. The idea right now is that the former is the one chosen as a good-practice standard.
- You have defined `acc`. Type `ACC` and see what happens when you try to work with it.

1.3 Simple types of objects

When programming (not only with R but in general) we will use elements such as `3`, `-2.5`, `"cool_element"`, `a_function_with_nice_properties()`... They are different in many ways but they all are objects for R. R understands that both `3` and `"cool_element"` are *things* with different properties. For instance, you can add `3` and `2` but can't `3` and `"cool_element"`. These properties the *thing* (or object) has are what make it belong to a class or other. `3` in the real life is a number and `"cool_element"` is text. R needs to know that so that

it will work differently with both of them. To do this, `3` belongs to the class `numeric` and `"cool_element"` to the class `character` (commonly called string in other languages). We'll get into details in the next lines.

When we want to know what type of object we're working with we'll use the function `class()`. Let's get into details.

Remark. When the documentation about a funcion is needed, is easily accesible via `? name_of_the_function`, e.g., `? class`. The focus of RStudio will move to the tab *Help*, on the lower left side of the window, by default.

1.3.1 Numeric

We have already seen some examples with numbers in R.

```
1
2.3
-1003.65
pi
```

1.3.1.1 Integer

Remember from high school the differences between real and integer numbers. 1, 1024, -53 are integers, but 0.5, -1.333 or π aren't. In R there exists a way of indicating such a difference and is important when dealing with large sets of data. The reason behind this is that R needs less information about an integer than about a real number. We will not get into the technical details in this course.

If we define a variable `my_number` as `my_number <- 10`, R will not understand the variable as integer, but a real number, therefore, numeric.

```
my_number <- 10
class(my_number)
```

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

For letting R know this variable should be define as integer, we use `L`.

```
my_number <- 10L
class(my_number)
```

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

Numerical operations apply similarly to integer variables.

1.3.2 Character

Even though statistics deals always with numbers, saving text data is mandatory to understand what information we have. For example, we may have a table with population data of several cities, but we need to have those cities' names to relate each piece of population data with the city. Thus, we need a way to tell R that a text is a text, so that it will understand it doesn't have to work with it as if it were a number.

The way of doing this is `""`. Some examples:

```
city <- "Madrid"
capital_letters <- "NEW YORK"
a_number <- "10"
strange_characters <- "@!-/"
all_together <- "¿Cuál es la capital de España?"
```

```
class(city)
```

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

```
class(all_together)
```

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

We can also operate with character (or string) objects, generally with functions. We may see some examples in other sections but it is not a main goal of the course.

1.3.3 Logical

When programming, you will need to tell the computer to do something taking into account a condition. For example, creating a new folder if it doesn't exists yet. Thus, R must know whether something is true or false. For this, we have the reserved words `TRUE` and `FALSE` (capital letters).

This is crucial when developing your own processes and functions but for now it'll be enough getting the general idea.

```
saying_yes <- TRUE
class(saying_yes)
```

```
## [1] "logical"
```

Remark. Generally, the logical type is called boolean in other programming languages and it is easy finding documentation refering to this word, even about R.

Another remark. What the programming language hide behind a logical value is a number. `TRUE` equals 1 and `FALSE` equals 0.

We have seen some examples of operations with numbers but we can also compare them in the usual way.

```
pi > 3
```

```
## [1] TRUE
```

```
log10(100) == 2
```

```
## [1] TRUE
```

```
-3 <= -4
```

```
## [1] FALSE
```

```
TRUE == 1
```

```
## [1] TRUE
```

```
FALSE == 0
```

```
## [1] TRUE
```

```
TRUE == 3
```

```
## [1] FALSE
```

We can assign these comparisons to variables.

```
comparison1 <- pi > 3
comparison2 <- log10(100) == 2
comparison3 <- -3 <= -4

class(comparison1)
```

```
## [1] "logical"
```

```
class(comparison2)
```

```
## [1] "logical"
```

```
class(comparison3)
```

```
## [1] "logical"
```

We will see a practical example of this when introducing the tables (data frames).

1.4 Sets of elements

1.4.1 Vectors

In statistics we usually work with sets of data, i.e., several numbers. Imagine we have data about the height of several people and want to calculate the average height. We can pass this numbers to the R function `mean()`, which will return the desire result.

```
mean(1.75, 1.69, 1.81, 1.65, 2.01, 1.73, 1.90, 1.62)
```

```
## [1] 1.75
```

If we want to use that set again for another calculations (e.g., standard deviation with `sd()`), we need to define a new variable in R with those numbers. Writing them in the same way we have just done would return an error.

```
my_set <- 1.75, 1.69, 1.81, 1.65, 2.01, 1.73, 1.90, 1.62
# Error: unexpected ',' in "my_set <- 1.75,"
```

For managing R to understand those numbers should be saved in the same variable all together, with need the `c()` function.

```
my_set <- c(1.75, 1.69, 1.81, 1.65, 2.01, 1.73, 1.90, 1.62)
my_set
```

```
## [1] 1.75 1.69 1.81 1.65 2.01 1.73 1.90 1.62
```

```
mean(my_set)
```

```
## [1] 1.77
```

```
sd(my_set)
```

```
## [1] 0.1316923
```

The avarage height is 1.77 meters and the standard deviation is 13 centimeters.

These sets of elements are called vectors. A vector in R is a set of objects of the **same** type. However, if ask R about the class of a vector, it will say the class is the one of the elements.

```
class(c(2, 4, 6))
```

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

```
class(c("hi there!", "I don't really understand this R stuff", "abc10"))
```

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

```
class(c(TRUE, TRUE, FALSE, TRUE, FALSE))
```

```
## [1] "logical"
```

The length of the vector is the number of elements it contains and can be consuted with the `length()` function. We can extract individual elements from a vector or subsets of it with squared brackets (`[]`) and the desired indexes. **The index of the first element of the vector is 1.** For several programming languages, such as Python the first index is 0.

```
my_set[2]
```

```
## [1] 1.69
```

```
my_desired_index <- 3
my_set[my_desired_index]
```

```
## [1] 1.81
```

Be careful with the notation when mixing vector and functions.

```
my_set[length(my_set)]
```

```
## [1] 1.62
```

There is no element in the vector whose index is 0.

```
my_set[0]
```

```
## numeric(0)
```

For subsetting instead of extracting indivual elements, we need to provided the `[]` with another vector, whose numbers will be the indexes of the elements we want.

```
my_indexes <- c(2, 4, 6)
my_set[my_indexes]
```

```
## [1] 1.69 1.65 1.73
```

```
my_set[-my_indexes] ## all except my_indexes
```

```
## [1] 1.75 1.81 2.01 1.90 1.62
```

For simplyfing, there is no need on writing `c(1, 2, 3, 4, 5)`. You can use the sequence notation `1:5`.

```
a <- 1:5
b <- c(1, 2, 3, 4, 5)

a == b ## element by element
```

```
## [1] TRUE TRUE TRUE TRUE TRUE
```

```
a
```

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

Exercise. What happens when you define a variable with the name `c`? Bear in mind that `c()` is a function in R. *Spoiler alert.* Not much: you just need to be careful and remember you have two different things with the same name, but there wil be no ambiguity since the uses of one of them are different form the other's.

You can also use this `:` notation for indexes, as well as logical vectors and integers variables.

```
my_set[4L]
```

```
## [1] 1.65
```

```
logical_vector <- c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)
my_set[logical_vector]
```

```
## [1] 1.75 1.81 2.01 1.90
```

Notice that, in the logical indexes case, the length of the index vector and the main vector are the same. This doesn't have to be like this always, since R knows how to recycle information, though it may be counterintuitive is some ocasions.

```
my_set[c(TRUE, FALSE)]
```

```
## [1] 1.75 1.81 2.01 1.90
```

```
my_set[c(TRUE, FALSE, TRUE)]
```

```
## [1] 1.75 1.81 1.65 1.73 1.90
```

1.4.2 Matrix

You may have studied matrixes back in high school but for us we will understand them as tables with elements, distributed along rows and columns. As well as we saw with vectors, the elements of a matrix must belong all to the same class.

Vectors were defined via `c()` but there is no way of indicating a second dimension to this function, so that we can have rows and columns. For achieving this, we use `matrix()`.

```
my_matrix <- matrix(c(10, 12, 9, 1, 5, 7, -1, -6, 8, 100, 200, 300), nrow = 3)
my_matrix
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10    1   -1  100
## [2,]   12    5   -6  200
## [3,]    9    7    8  300
```

There are several considerations that must be taken into account when defining matrixes but we won't get into them in this course since we will focus on data frames. We'll just mention a couple of thigs related with working already defined matrixes.

A vector has one only dimension and the number of elements is its length. For matrixes, there is no length concept. Instead, we work with rows and columns.

```
nrow(my_matrix)
```

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

```
ncol(my_matrix)
```

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

We have tow dimensions, therefore we need to specify two indexes when extracting elements.

```
desired_row <- 2
desired_column <- 4

my_matrix[2, 4]
```

```
## [1] 200
```

Of course, vectors can also be used as indexes for submatrixes.

```
my_matrix[1:2, 3:4]
```

```
##      [,1] [,2]
## [1,]   -1  100
## [2,]   -6  200
```

1.4.3 Data frames

In statistics, a popular format for keeping data are tables. In R, the main type of tables is called data frame. A data frame is a table, as well as matrixes, but its elements don't have to be all of the same type. The first column can contain integers, the second, characters, the third, numbers, the fourth, logicals, and so on.

```
my_data_frame <- data.frame(
  col1 = 1:4,
  col2 = c("Category 1", "Category 2", "Category 3", "Category 4"),
  col3 = c(3.4, -2.5, 10.1, -0.05),
  col4 = c(T, F, F, T)
)

my_data_frame
```

```
##   col1       col2  col3  col4
## 1    1 Category 1  3.40  TRUE
## 2    2 Category 2 -2.50 FALSE
## 3    3 Category 3 10.10 FALSE
## 4    4 Category 4 -0.05  TRUE
```

Exercises.

- Check the type of each column for the data frame you've just created. Remember you need the function `class()` to do it.
- Is there anything strange with the second column? What were you expecting? Ask your teach about this and feel free to read `? factor` but don't worry about it for now. We'll get into that later.

Data frames have some things in common with matrixes. They are both tables, therefore the same sintax can be used to get their elements:

```
my_data_frame[1, 2] # first row, second column
```

```
## [1] "Category 1"
```

```
my_data_frame[2:4, 2] # second to fourth rows, second column
```

```
## [1] "Category 2" "Category 3" "Category 4"
```

```
my_data_frame[1:3, -4] # first to third rows (included), all except fourth column
```

```
##   col1       col2 col3
## 1    1 Category 1  3.4
## 2    2 Category 2 -2.5
## 3    3 Category 3 10.1
```

Exercise. What are the types of objects returned with those executions.

As you may have realized, this is a bit messy for beginning working with R. We have mentioned it because it is important to know that this exist, but we will not work with data frames as matrix, but we'll use the proper tools for data frames that R provides.

First of all, we need to understand how to extract one column in the form of a vector.

1.4.3.1 `$` and `[[`

You may have noticed that we defined our dataframe with named columns: col1, col2 and so on (it is worth mentioning that this names can be whatever you desire, following some rules we'll not describe now). This is very convenient since it makes easy getting the elements from each column:

```
my_data_frame$col1
```

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

```
my_data_frame[["col1"]]
```

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

- `$` has the advantage of autocompletion (with Tab).
- `[[` has the advantage of parametrizing via characters, something very useful when dealing with functions, mainly by the end of the course.

There are several ways of verifying those vector are equal. We choose this one:

```
dollar_sintax <- my_data_frame$col1
double_bracket_sintax <- my_data_frame[["col1"]]

all(dollar_sintax == double_bracket_sintax)
```

```
## [1] TRUE
```

we could have done it at a glance but if you have a one million column data frame, this way's better ;)

Remark. Notice we typed double squared bracket (`[[`), not single (`[`). The character notation can also be used in the second case but differently and there are more cases to be considered. For simplicity's sake, we won't get into that.

We don't know yet how to read data from files (patience, my dear reader) but R makes comfortable using some preloaded datasets as examples. A classical one is *iris*, a dataset with a some of flowers of three species and some info about them. We will use as a example for a while.

First we should have an idea about the general structure of the data frame. We can type `iris` on the console and see the data but it is not very clarifying, since there are too many data and don't fit tidily inside the tab.

A common function for a first approach is `head()`.

```
head(iris)
```

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

Cool. `head()` shows by default the first six rows of the data frame used as input (be careful if you have too many columns). The analogous is `tail()`, which shows the last six rows. This number is a parameter of the function and can be redefined.

```
tail(iris, 10)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 141          6.7         3.1          5.6         2.4 virginica
## 142          6.9         3.1          5.1         2.3 virginica
## 143          5.8         2.7          5.1         1.9 virginica
## 144          6.8         3.2          5.9         2.3 virginica
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

For checking the number of rows and columns of the data frame (these ones also work with matrixes):

```
nrow(iris)
```

```
## [1] 150
```

```
ncol(iris)
```

```
## [1] 5
```

```
dim(iris)
```

```
## [1] 150   5
```

We have easily verified there are 150 rows and 5 columns. If we want to know what columns there are, we can get their names with `names()`:

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

We have info about the sepal, the petal and the species each flower belongs to. We could have also used `colnames()` for this. Bear in mind that this one would also work with matrixes but the former wouldn't.

We'll come back later to have a look at the structure of a data frame with new tools. For now, let's go on with the `$` notation.

`iris` is a data frame.

```
class(iris)
```

```
## [1] "data.frame"
```

But its columns, from an independent point of view, are vectors. Therefore, their classes are the ones of their elements.

```
class(iris$Sepal.Length)
```

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

```
class(iris$Sepal.Width)
```

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

```
class(iris$Petal.Length)
```

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

```
class(iris$Petal.Width)
```

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

```
class(iris$Species)
```

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

We have some numbers and characters (again, don't worry about the *factor* thing yet. We'll come to that). A typical function to get a general idea of the distributions of the columns is `summary()`.

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

Let's count the number of flowers with a longer sepal than the mean (5.8, according to the summary).

- First we need to know, for each row, whether the sepal length is greater than 5.8: `iris$Sepal.Length > 5.8`.

- That piece of code will return a vector of 150 `TRUE` or `FALSE` values, indicating what we want to know.
- Now we need to count how many trues there are. But remember that `TRUE == 1`, so a good way to counting is adding all the `TRUE` cases, with `sum()`.

```
sum(iris$Sepal.Length > 5.8)
```

```
## [1] 70
```

There is no need in typing the mean manually. You can use the function `mean()` instead.

```
my_average <- mean(iris$Sepal.Length)
sum(iris$Sepal.Length > my_average)
```

```
## [1] 70
```

This is an example of comparing every row with the same number. But you can use different values for each row. For example, let's count the number of flowers whose sepal is longer than its petal.

```
sum(iris$Sepal.Length > iris$Petal.Length)
```

```
## [1] 150
```

All of them. Naïve example.

We can also use these logical vectors to extract subvectors with information. For instance, let's print the sepal length of the flowers whose petal width is above the median:

```
my_index <- iris$Petal.Width > median(iris$Petal.Width) # is above the median

head(my_index) # these are logical values indicating whether the flower corresponds with the fil
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
iris$Sepal.Length[my_index]
```

```
##  [1] 7.0 6.4 6.9 6.5 6.3 5.2 5.9 6.1 6.7 5.6 6.2 5.9 6.3 6.6 6.8 6.7 6.0 6.0 5.4
## [20] 6.0 6.7 6.1 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4
## [39] 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1
## [58] 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

Now we have a subvector of the sepal length. Let's count the number of flowers with this caracteristic. This will be the number of elements in the vector, what we can calculate with `length()`.

```
length(iris$Sepal.Length[my_index] )
```

```
## [1] 72
```

There are 72 flowers whose petal width is above the median.

Exercises.

- Count the number of flowers with Petal.Length higher than 5.1

```
sum(iris$Petal.Length > 5.1)
```

```
## [1] 34
```

- Check which species have flowers with sepal width higher than 3.5 and lower than 4. *Hint.* `&` will be needed when more than one condition is needed. Besides, for simplifying the result you may need `unique()`. Use `?` for details.

```
unique(iris$Species[iris$Sepal.Width > 3.5 & iris$Sepal.Width < 4])
```

```
## [1] setosa    virginica
## Levels: setosa versicolor virginica
```

- How many flowers of each species are there? *Hint.* Function `table()` may be useful.

```
table(iris$Species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

- Calculate the sepal width average of the flowers whose sepal width is below average.

```
mean(iris$Sepal.Length[iris$Sepal.Width < mean(iris$Sepal.Width)])
```

```
## [1] 5.972289
```

- Calculate the median of the petal width for each species separetely.

```
print("Setosa:")
```

```
## [1] "Setosa:"
```

```
median(iris$Petal.Width[iris$Species == "setosa"])
```

```
## [1] 0.2
```

```
print("Versicolor:")
```

```
## [1] "Versicolor:"
```

```
median(iris$Petal.Width[iris$Species == "versicolor"])
```

```
## [1] 1.3
```

```
print("Virginica:")
```

```
## [1] "Virginica:"
```

```
median(iris$Petal.Width[iris$Species == "virginica"])
```

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

- Calculate the minimum and maximum petal length for every versicolor flower. Then count how many setosa and virginica flowers there are with the petal length between those values.

```
print("Min:")
```

```
## [1] "Min:"
```

```
minimo_valor <- min(iris$Petal.Length[iris$Species == "versicolor"])
minimo_valor
```

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

```
print("Max:")
```

```
## [1] "Max:"
```

```
maximo_valor <- max(iris$Petal.Length[iris$Species == "versicolor"])
maximo_valor
```

```
## [1] 5.1
```

```
print("Versicolor:")
```

```
## [1] "Versicolor:"
```

```
especie <- "setosa"
logical_for_especie <- iris$Species == especie
petal_lengths_for_especie <- iris$Petal.Length[logical_for_especie]

true_si_mayor_que_min <- petal_lengths_for_especie > minimo_valor
true_si_menor_que_max <- petal_lengths_for_especie < maximo_valor

# Casos que cumplen ambas cosas:
# Esto es un vector lógico: true_si_mayor_que_min & true_si_menor_que_max
# Puedo sumar los valores para hacer el conteo:
sum(true_si_mayor_que_min & true_si_menor_que_max)
```

```
## [1] 0
```

```
print("Virginica:")
```

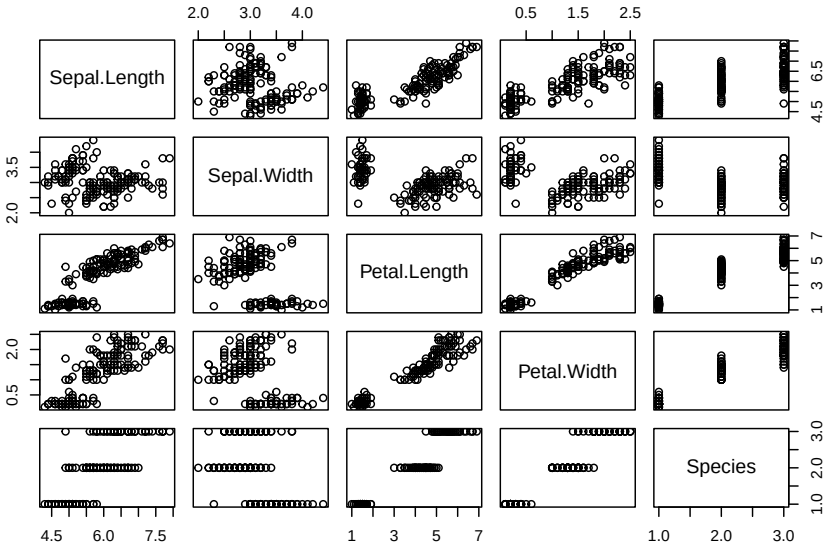
```
## [1] "Virginica:"
```

```
especie <- "virginica"
logical_for_especie <- iris$Species == especie
petal_lengths_for_especie <- iris$Petal.Length[logical_for_especie]

true_si_mayor_que_min <- petal_lengths_for_especie > minimo_valor
true_si_menor_que_max <- petal_lengths_for_especie < maximo_valor

sum(true_si_mayor_que_min & true_si_menor_que_max)
```

```
## [1] 9
```

- What happens if you use `plot()` with the data frame `iris`?

```
plot(iris)
```



- If you type `Titanic` on the console you will set a data set with a not too comfortable format for analysis. Use `as.data.frame()` to convert it to a data frame and assign it to a new variable (object) in R (pick a name of your choice). Now that data frame is on memory.

```
df_titanic <- as.data.frame(Titanic)
```

- Verify that data frame is in fact a data frame.

```
class(df_titanic)
```

```
## [1] "data.frame"
```

- Check the classes of the data frame's columns

```
class(df_titanic$Class)
```

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

```
class(df_titanic$Sex)
```

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

```
class(df_titanic$Age)
```

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

```
class(df_titanic$Survived)
```

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

```
class(df_titanic$Freq)
```

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

- Count (or sum) the number of people who survived and the number of people who didn't. If you have questions about the data, type `? Titanic` to get into the help (yes! You also have help for datasets.).

```
print("Survived:")
```

```
## [1] "Survived:"
```

```
sum(df_titanic$Freq[df_titanic$Survived == "Yes"])
```

```
## [1] 711
```

```
print("Didn't survive")
```

```
## [1] "Didn't survive"
```

```
sum(df_titanic$Freq[df_titanic$Survived == "No"])
```

```
## [1] 1490
```

- Count the number of children who were on the ship.

```
sum(df_titanic$Freq[df_titanic$Age == "Child"])
```

```
## [1] 109
```

- Count the number of men from first class.

```
## [1] 180
```

- Were there any children among the crew? And women?

```
print("Children:")
```

```
## [1] "Children:"
```

```
sum(df_titanic$Freq[df_titanic$Age == "Child" & df_titanic$Class == "Crew"])
```

```
## [1] 0
```

```
print("Women:")
```

```
## [1] "Women:"
```

```
sum(df_titanic$Freq[df_titanic$Sex == "Female" & df_titanic$Class == "Crew"])
```

```
## [1] 23
```

1.5 Extending R with libraries

What we've seen so far belongs to what it is known as R base. When programming in R, we'll use a set of words and symbols that the computer understands and operates according to them. But because of the complexity of the projects

that are usually developed in Data Science, programming everything in R base may become verbose, difficult and the least practical way of working.

Several people from the R community work continously developing sets of code that, with new words and symbols, expand R and its uses. These sets of code are called libraries or packages (not only in R but in every programming language, such as Python).

For using a library, we employ the function `library()`, with the name of the desired library as the parameter.

For instance, we have seen the `data.frame` structure. If we want to build a data frame from scratch we will type `data.frame()`, with some data between the brackets. However, there are other types of *data frames* in R, which are more efficient from a couple of point of views (we will explain these during the sessions, probably). In this course, we are going to use one of these types: the *tibble*.

Tibbles are redefined *data frames* and the way of building one is similar: we type `tibble()`, with the data in form of vectors inside the brackets. The problem is that if we type that directly, we'll get an error:

```
tibble(
  my_cool_column = c(23, 56, 77),
  my_lovely_column = c("thing1", "thing2", "thing3")
)

# Error...: could not find function "tibble"
```

R is telling us that there is no such a function `tibble()`. In fact, there isn't... at least in R base. There is one `tibble()` function in some other place: in one library (or several, but we'll get into that later on).

```
library(tibble)

tibble(
  my_cool_column = c(23, 56, 77),
  my_lovely_column = c("thing1", "thing2", "thing3")
)
```

```
## # A tibble: 3 x 2
##   my_cool_column my_lovely_column
##            <dbl> <chr>
## 1             23 thing1
## 2             56 thing2
## 3             77 thing3
```

Now it works. We'll study tibbles during the sessions. The important thing to be remembered R base will not be a comfortable tool for our data science projects alone: we will need it expanded. And the way for achieving this are libraries.

1.6 Installation

Remark. The code above may not have worked in your local computer. **You need to install a library before using it.**

Installing libraries (or packages: two words, same thing) is required in order to make use of them. In general, all we need to do is typing `install.package()`. Between the brackets we will include the name of library with quotation marks.

```
install.packages("tibble")
```

Recommendation. Once you install a library on your computer, there is no need to do it again. It is strongly recommended to type the installation command on the console, not on your scripts, so that the library won't be reinstalled all over again.

Exercise. During the session, if you pay attention, we will talk about the *tidyverse*. It is a set of libraries, containing the two most important for our course: dplyr and ggplot2. Installing tidyverse, dplyr, ggplot2, tibble, readr and a lot more will also be intalled along. Install tidyverse on your computer. Remember to use the console. *Hint.* When you install a library, a lot of messages will come up on the console. Don't worry: they are only messages, not errors. If you really read the word `Error` on the console, ask your teacher.

Chapter 2

Reading data with readr

2.1 Excel

Unfortunately, Excel is used worldwide in a lot of sectors and along with several technologies to store and analyze data. But it has limitations in scalability, generalization, license... R is a solution for all this. If you need to analyse some data stored in Excel, you can read it from R and start working with this programming language.

In the `data` folder you have the `iris` dataset exported as an Excel file (watch out! you also have it with a different format, but will not be used yet). You can open it with Excel or other spreadsheet software such as Numbers, LibreOffice or Google Sheets. To work with it in R, we need to load it on memory as a data frame or tibble (we are going to work with the latter). There is no way of doing this with R base, but there are a few options on different libraries. For reading, we'll use `readxl`.

```
library(readxl)
iris_excel <- read_excel("data/iris.xlsx")
```

The function `read_excel()` receives a path in your computer to the file you want to read. We will explain in class the concept of relative and absolute path (in the example, the path is relative, since we have read a file inside the working directory, which you can verify with `getwd()`).

We can print `iris_excel` on the console and check that the info is an already known tibble.

```
iris_excel
```

```
## # A tibble: 150 x 5
##    sepal_length sepal_width petal_length petal_width species
##           <dbl>       <dbl>        <dbl>       <dbl> <chr>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

You have been provided with another Excel file, `"01_ ACCIDENTES POR TIPO EN  DISTRITOS.xls"`. If you open it, you will see that there are several sheets and that the data don't begin on the first row. If you use the code above to read it into R, you will not get a data frame as you would like.

```
datos_mal <- read_excel("data/01_ ACCIDENTES POR TIPO EN  DISTRITOS.xls")
```

```
## New names:
## * `` -> ...2
## * `` -> ...3
## * `` -> ...4
## * `` -> ...5
## * `` -> ...6
## * ...
```

```
datos_mal
```

```
## # A tibble: 29 x 12
##    `01. ACCIDENTES ~ ...2  ...3  ...4  ...5  ...6  ...7  ...8  ...9  ...10 ...11
##    <chr>             <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
##  1 <NA>              <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
##  2 <NA>              <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
##  3 <NA>              <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
##  4 <NA>              <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
##  5 Indicadores       Nº A~ <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
##  6 Año               2009  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>  <NA>
##  7 DISTRITO_ACCIDEN~ COLI~ COLI~ CHOQ~ ATRO~ VUEL~ CAÍD~ CAÍD~ CAÍD~ CAÍD~ OTRA~
##  8 ARGANZUELA        389   54    126   75    9     43    9     4     5     8
##  9 BARAJAS           89    6     53    21    <NA>  12    2     1     2     2
```

```
## 10 CARABANCHEL       375   44    171   137   8     36    13    6     9     8
## # ... with 19 more rows, and 1 more variable: ...12 <chr>
```

We need to consider two parameters in `read_excel()` function: the row number where we want to begin the reading and the sheet we want to read.

```
df_accis_2009 <- read_excel("data/01_ ACCIDENTES POR TIPO EN  DISTRITOS.xls",
                            skip = 7, sheet = "2009")
```

```
## New names:
## * `` -> ...12
```

Exercises.

- Read the rest of the sheets of the Excel file. You should find some trouble with one of the years if you just copy and paste what I typed. Fix it.
- Create one data frame for each sheet. How many rows and columns each one has? *Hint.* `nrow()`, `ncol()` and `dim()` may help.
- What is the average number of accidents per type in 2009? And the standard deviation? *Hint.* You should use the `$` notation.
- What are the names of the columns? Do you consider them appropriate? *Hint.* `names()`.

2.1.1 About the names

In R there is some freedom about the names of the columns of a data frame but it is recommeded some guidance, similar to the one of creating objects:

- Use only letters (I only use lowercase), numbers and `_`.
- Separate each word in the name with a `_`.
- Begin only with letters.

There is a function that helps a lot fixing the names of columns. It is especially orientated to problems related with Excel files. The function is `clean_names()` and you can find it in the `janitor` package.

Exercise.

- Install the `janitor` package and load it. *Hint.* `library()`.
- Use `clean_names()` on your data frames to fix their names. *Hint.* Bear in mind that if you just type `clean_names(mi_data_frame_guay_recien_creado)`, R will show you on the console the result, but the names won't be overwritten. You need to assign that result to the data frame with `<-`.

2.2 What is a csv file

Excel is commonly used for working with tabular data. Excel files allow not only keeping data but also working with them, with formulas and plots. However, when working with too many rows or columns becomes herculean or even impossible (today, Excel doesn't even allow two million rows). When dealing with these not so large datasets, other formats will be needed for saving the data. However, while Excel has a set of tools for data analysis, these alternatives are only for saving data; for analyzing it, we need R (or other programming language and data software).

Structure data is usually stored in a tabular format, i.e., in tables. One of the most commong tabular formats is csv (comma separated values). During the sessions we'll get into details on what a csv file looks like but, from a general point of view, what we need to know is that it is plain text: you can generate a csv file from scratch with the notepad.

Imagine you have the already known `iris` dataset on a csv file (this file will have a name, as every file, but the extension will be `.csv`). It will look something like this:

```
Sepal.Length,Sepal.With,Petal.Length,Petal.Width,Species
5.1,3.5,1.4,0.2,setosa
4.9,3.0,1.4,0.2,setosa
4.7,3.2,1.3,0.2,setosa
```

Since it is not aligned, it is not intuitive realizing that we have three rows of the dataset. There are four rows but the first one is the titles of the columns. Each column is separated with a comma (this is the reason of the name of the format). For every new row, we press Enter at the end of the line.

We will now get into how to read this with R, so that such a file on our hard drive can be read onto memory (from disk to RAM, which is where R *almost always* works).

2.3 Reading a csv

On the `data` folder you have another file with the `iris` dataset, this time in csv format.

There are some functions in R base for reading csv files but we won't use them. We will apply the `read_csv()`, from the `readr` package.

```
library(readr)
iris_csv <- read_csv("data/iris.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   sepal_length = col_double(),
##   sepal_width = col_double(),
##   petal_length = col_double(),
##   petal_width = col_double(),
##   species = col_character()
## )
```

Exercise.

- How many columns are there on the tibble?
- How many rows?
- What are the types of the columns? *Hint.* Use `class()`.

Depending on your computer's configuration, if you export a file from Excel to csv, the columns may be separated by commas or semicolon, and the decimal mark for numbers may be a point or a comma, respectively.

Exercise. - Read the `iris2.csv` file with the `read_csv()` function. - How many columns are there? Which are their names? Does it make any sense?

2.4 Reading other tabular data

2.4.1 Another csv

Not always the character for separating columns will be a comma. In some countries, Spain for example, because of the format we usually use for numbers, instead of separating the columns of a csv file with commas, we use semicolons. Thus, we avoid issues with decimal numbers, since we use a comma for separating the integer and decimal part (in other countries, a point is used).

This file format is also called csv, but an alternative on the code must be typed:

```
iris_csv <- read_csv2("data/iris2.csv")
```

```
## i Using "','" as decimal and "'.'" as grouping mark. Use `read_delim()` for more control.
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   Sepal.Length = col_double(),
```

```
##   Sepal.Width = col_double(),
##   Petal.Length = col_double(),
##   Petal.Width = col_double(),
##   Species = col_character()
## )
```

2.4.2 tsv

Another typical way of separating columns is using tab. It is not exactly a space for the computer, though our eyes may no detect many differences. The extension for these files usually is `.tsv`..

In `readr` we can use `read_tsv()` function for this format.

```
iris_tsv <- read_tsv("data/iris.tsv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   Sepal.Length = col_double(),
##   Sepal.Width = col_double(),
##   Petal.Length = col_double(),
##   Petal.Width = col_double(),
##   Species = col_character()
## )
```

2.4.3 Release your imagination

You have complete freedom when choosing a character for separating columns. A very typical one is `|`. For these alternatives, there is a general function in the `readr` package that allows you manually select the character that should be used: `read_delim()`. This function has two mandatory parameters: the path where you have the file stored and the character used for separations.

Remark. For this file, the extension will usually be `.txt`, but you can invent it (e.g., `.sal`, for output in Spanish) or even nothing.

```
iris_nuevo <- read_delim(file = "data/iris.txt", delim = "|")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   Sepal.Length = col_double(),
##   Sepal.Width = col_double(),
```

```
##   Petal.Length = col_double(),
##   Petal.Width = col_double(),
##   Species = col_character()
## )
```

Exercise. On the `data` folder there is a file with no extension. Figure out the character used of separating the columns and read it.

2.5 More things

Files don't have to be on your local computer always. You can also read files stored somewhere on the web.

```
read_csv("https://github.com/tidyverse/readr/raw/master/inst/extdata/mtcars.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   mpg = col_double(),
##   cyl = col_double(),
##   disp = col_double(),
##   hp = col_double(),
##   drat = col_double(),
##   wt = col_double(),
##   qsec = col_double(),
##   vs = col_double(),
##   am = col_double(),
##   gear = col_double(),
##   carb = col_double()
## )
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
##  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
```

```
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
## # ... with 22 more rows
```

This dataset is very common in data analysis courses, such as *iris*. It contains information about a set of cars. You can read the details in the documentation: `? mtcars`.

Exercise. Check the classes of all the columns.

Some columns have been considered by R as numeric but don't have decimal numbers. We should convert them to integer, since this type of object needs less space in memory than real numbers. We don't know yet how to change the columns of a data frame or tibble, but we can do these changed directly when reading the file.

All the family of functions `read_*()` from the `readr` package guess the types of the columns (this may be explained during the sessions) but there is a way of forcing it: the `col_types` parameter.

There are a couple of ways of specifying these types. Here is an example for one. We use a character vector with one letter for each columns. The letters will represent the types: in this case, `n` for `numerical`, `i` for `integer` and `l` for `logical`.

```
read_csv("https://github.com/tidyverse/readr/raw/master/inst/extdata/mtcars.csv", col_t
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs am     gear  carb
##    <dbl> <int> <dbl> <dbl> <dbl> <dbl> <dbl> <int> <lgl> <int> <int>
##  1  21       6  160    110  3.9   2.62  16.5     0 TRUE      4     4
##  2  21       6  160    110  3.9   2.87  17.0     0 TRUE      4     4
##  3  22.8     4  108     93  3.85  2.32  18.6     1 TRUE      4     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1 FALSE     3     1
##  5  18.7     8  360    175  3.15  3.44  17.0     0 FALSE     3     2
##  6  18.1     6  225    105  2.76  3.46  20.2     1 FALSE     3     1
##  7  14.3     8  360    245  3.21  3.57  15.8     0 FALSE     3     4
##  8  24.4     4  147.    62  3.69  3.19  20       1 FALSE     4     2
##  9  22.8     4  141.    95  3.92  3.15  22.9     1 FALSE     4     2
## 10  19.2     6  168.   123  3.92  3.44  18.3     1 FALSE     4     4
## # ... with 22 more rows
```

Chapter 3

Working with tables

The main library we'll use is dplyr. We are also using readr for reading files onto R.

```
library(dplyr)
library(readr)
```

3.1 Creating and selecting columns

3.1.1 mtcars dataset

Let us begin creating columns or redefining existing ones. For this, we use the `mutate()` function. It receives a data frame or tibble (mainly via the *pipe* operator, `%>%`) and performs a formula for calculating new or existing columns.

We read the data with the `read_csv()` function from the `readr` package.

```
df_mtcars <- read_csv("data/mtcars.csv")
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   mpg = col_double(),
##   cyl = col_double(),
##   disp = col_double(),
##   hp = col_double(),
##   drat = col_double(),
##   wt = col_double(),
```

```
##   qsec = col_double(),
##   vs = col_double(),
##   am = col_double(),
##   gear = col_double(),
##   carb = col_double()
## )
```

The data has been obtained from R base and you can read the documentation with `? mtcars`.

`read_csv()` gives us some information about the reading. We can see that all columns belong to class `numeric`. We can have a more detailed look at the tibble with `glimpse()`, which shows us the first rows and some info.

```
df_mtcars %>% glimpse()
```

```
## Rows: 32
## Columns: 11
## $ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8,~
## $ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, 4, 4, 8,~
## $ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140.8, 16~
## $ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 180, 180~
## $ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92, 3.92,~
## $ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.150, 3.~
## $ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.90, 18~
## $ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0,~
## $ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0,~
## $ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, 4, 3, 3,~
## $ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, 2,~
```

Exercise. What are the differences between `glimpse(df_mtcars)` and `df_mtcars %>% glimpse()`?

With the first rows we can confirm that all the columns are numerical but we can detect some differences among them. For instance, `cyl` and the last ones are integer numbers, so they can represent countings, or even categories, since values at `vs` or `am` seem only 0 and 1.

We can have more numerical information with `summary()`, from R base.

```
summary(df_mtcars)
```

```
##       mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
```

```
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
##       drat             wt             qsec             vs
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
##        am              gear            carb
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
##  Median :0.0000   Median :4.000   Median :2.000
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

For calculating new or existing rows we use `mutate()`, as mentiobed before. If we only use `mutate()`, R shows us the data frame with the new calculations on the console, but does not overwrite it. We need to assing the result to the data frame to overwrite the result.

For intance, the weight is given in 1000 lbs. Let's change the units.

First, we can have a look at the first rows of this column with the function `select()`.

```
df_mtcars %>%
  select(wt)
```

```
## # A tibble: 32 x 1
##       wt
##    <dbl>
##  1  2.62
##  2  2.88
##  3  2.32
##  4  3.22
##  5  3.44
##  6  3.46
##  7  3.57
##  8  3.19
##  9  3.15
## 10  3.44
## # ... with 22 more rows
```

Now let's make the calculation.

```
df_mtcars %>%
  mutate(wt = wt * 1000)
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  21       6  160    110  3.9   2620  16.5     0     1     4     4
##  2  21       6  160    110  3.9   2875  17.0     0     1     4     4
##  3  22.8     4  108     93  3.85  2320  18.6     1     1     4     1
##  4  21.4     6  258    110  3.08  3215  19.4     1     0     3     1
##  5  18.7     8  360    175  3.15  3440  17.0     0     0     3     2
##  6  18.1     6  225    105  2.76  3460  20.2     1     0     3     1
##  7  14.3     8  360    245  3.21  3570  15.8     0     0     3     4
##  8  24.4     4  147.    62  3.69  3190  20       1     0     4     2
##  9  22.8     4  141.    95  3.92  3150  22.9     1     0     4     2
## 10  19.2     6  168.   123  3.92  3440  18.3     1     0     4     4
## # ... with 22 more rows
```

If there are too many columns is uncomfortable seeing the results. We can use `select()` and make the pipeline longer.

```
df_mtcars %>%
  mutate(wt = wt * 1000) %>%
  select(wt)
```

```
## # A tibble: 32 x 1
##       wt
##    <dbl>
##  1  2620
##  2  2875
##  3  2320
##  4  3215
##  5  3440
##  6  3460
##  7  3570
##  8  3190
##  9  3150
## 10  3440
## # ... with 22 more rows
```

The calculation seems correct. Now, instead of showing the result on the console, we overwrite the result (we don't use `select()` now because we want the entire tibble).

```
df_mtcars <- df_mtcars %>%
  mutate(wt = wt * 1000)
```

We don't see anything on the screen but R has changed the column `wt`. Had we made a mistake and wanted to go back, we would have to execute again the reading of the file or make the opposite calculation.

```
df_mtcars %>%
  mutate(wt = wt / 1000) %>%
  select(wt)
```

```
## # A tibble: 32 x 1
##       wt
##    <dbl>
##  1  2.62
##  2  2.88
##  3  2.32
##  4  3.22
##  5  3.44
##  6  3.46
##  7  3.57
##  8  3.19
##  9  3.15
## 10  3.44
## # ... with 22 more rows
```

The documentation explains that `vs` represents the shape of the engine. It is 0 or 1. Imagine you don't like this numbers and you want to use 1 and 2. You can easily add 1 to the column.

```
mtcars %>%
  mutate(vs = vs + 1) %>%
  select(vs)
```

```
##                     vs
## Mazda RX4            1
## Mazda RX4 Wag        1
## Datsun 710           2
## Hornet 4 Drive       2
## Hornet Sportabout    1
## Valiant              2
## Duster 360           1
## Merc 240D            2
## Merc 230             2
```

```
## Merc 280             2
## Merc 280C            2
## Merc 450SE           1
## Merc 450SL           1
## Merc 450SLC          1
## Cadillac Fleetwood   1
## Lincoln Continental  1
## Chrysler Imperial    1
## Fiat 128             2
## Honda Civic          2
## Toyota Corolla       2
## Toyota Corona        2
## Dodge Challenger     1
## AMC Javelin          1
## Camaro Z28           1
## Pontiac Firebird     1
## Fiat X1-9            2
## Porsche 914-2        1
## Lotus Europa         2
## Ford Pantera L       1
## Ferrari Dino         1
## Maserati Bora        1
## Volvo 142E           2
```

Remark. Remember to assign the result to the tibble with `<-` if you want to keep the results. Don't forget to erase `select()` in this case.

This change on the column may not seem very useful. It can be more interesting changing the numbers with the real meaning: V-shaped and straight. For this, we need the column to be a character, but it is numeric.

```
class(df_mtcars$vs)
```

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

We want to assign these meaning to the numbers and, therefore, the column will be a character, but R will take care of this for us. For changing the values, we use `if_else()`. We want to say: 'if vs equals 0, then it will equal "V-shaped", else "straight"'. We can create and auxiliar column `vs_aux` for making the change more clear.

```
df_mtcars %>%
  mutate(vs_aux = if_else(vs == 0, "V-shaped", "straight")) %>%
  select(vs, vs_aux)
```

```
## # A tibble: 32 x 2
##       vs vs_aux
##    <dbl> <chr>
##  1     0 V-shaped
##  2     0 V-shaped
##  3     1 straight
##  4     1 straight
##  5     0 V-shaped
##  6     1 straight
##  7     0 V-shaped
##  8     1 straight
##  9     1 straight
## 10     1 straight
## # ... with 22 more rows
```

Remark. Choose whatever names you want for the columns: it is not necessary to call the auxiliar one `vs_aux`. And you can overwrite the tibble or not: you are programming, you can always run everything again! =D

The idea of keeping the numbers instead of the character column is that numbers require less space on your computer's memory. So storing the shape with a numerical code is more efficient. In fact, you have already checked that the column is numerical but it would be better if it were integer.

Exercise. Decide what columns can be stored as integer without losing information and change them. For changing the type of an object or vector you can use `as.integer()`. For instance:

```
class(c(4, 6, 1))
```

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

```
class(as.integer(4, 6, 1))
```

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

```
df_mtcars %>%
  mutate(
    cyl = as.integer(cyl),
    vs = as.integer(vs),
    am = as.integer(am),
    gear = as.integer(gear),
    carb = as.integer(carb)
  )
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##    <dbl> <int> <dbl> <dbl> <dbl> <dbl> <dbl> <int> <int> <int> <int>
##  1  21       6  160    110  3.9   2620  16.5     0     1     4     4
##  2  21       6  160    110  3.9   2875  17.0     0     1     4     4
##  3  22.8     4  108     93  3.85  2320  18.6     1     1     4     1
##  4  21.4     6  258    110  3.08  3215  19.4     1     0     3     1
##  5  18.7     8  360    175  3.15  3440  17.0     0     0     3     2
##  6  18.1     6  225    105  2.76  3460  20.2     1     0     3     1
##  7  14.3     8  360    245  3.21  3570  15.8     0     0     3     4
##  8  24.4     4  147.    62  3.69  3190  20       1     0     4     2
##  9  22.8     4  141.    95  3.92  3150  22.9     1     0     4     2
## 10  19.2     6  168.   123  3.92  3440  18.3     1     0     4     4
## # ... with 22 more rows
```

3.1.2 Invented dataset

3.1.2.1 Why this is useful

Being able to simulate some random data is useful when you are developing some code but you haven't receive the final data yet. Nonetheless, because of the project's timing, you need to keep on working so that you'll have some code already prepared when you're data arrives and you don't need to begin from scratch by then.

3.1.2.2 Simulating data

This branch of Statistics takes several months at University to studying it. But for our purpose we just need some basics concepts.

Imaging you are betting on something and you need a coin. You can use a calculator or R to simulate the results. There is always some function that allows you to get a *random* number between 0 and 1 (I emphasized *random* because it requires some time to explain what that means, since nothing is random but controlled by the laws of physics, except maybe the location of an electron). Mathematically, you can use to simulate the results of a coin flip.

```
runif(10)
```

```
##  [1] 0.91291666 0.18087938 0.09463785 0.04986764 0.75348970 0.48415840
##  [7] 0.51100147 0.04810811 0.74593139 0.68888561
```

If you run that on your computer, you're results will be different (because they are random). Long story short, we can get the same numbers if we fix one thing called seed:

```
set.seed(1234)
runif(10)
```

```
##  [1] 0.113703411 0.622299405 0.609274733 0.623379442 0.860915384 0.640310605
##  [7] 0.009495756 0.232550506 0.666083758 0.514251141
```

Now you can say that lower than 0.5 results will represent tails and the rest, heads:

```
set.seed(1234)
results <- runif(10)

library(dplyr)
if_else(results < 0.5, "tails", "heads")
```

```
##  [1] "tails" "heads" "heads" "heads" "heads" "heads" "tails" "tails" "heads"
## [10] "heads"
```

3.1.2.3 Creating a data frame

You should know that what we have just created is a vector. And vectors can be used as columns on a data frame or tibble. Therefore, we'll invent some data following this methodology and take advantage of the data for some calculations.

Imagine we have 20 stores, with two dimensions, length and width, the number of customers we receive per day, the daily income and the colors of the walls (between green, red, blue and white (yes, wonderful colors for walls)). Let's simulate this.

For creating a tibble we use the function `tibble()`, from the tibble package but available on dplyr. Inside the function we will introduce the vectors with the simulations and the names of the columns. First, the vectors.

```
number_of_stores <- 20

indices <- 1:number_of_stores # Index: 1, 2, 3, 4, ..., 20

# For the random data we can do the seed thing so that the results will be the same
# for all of us
set.seed(2718)

length_sim <- rnorm(number_of_stores, mean = 7, sd = 1.5)
width_sim <- rnorm(number_of_stores, mean = 10, sd = 2.1)
```

```r
# For the customers, we assume that the average will be 50.
# You'll learn what a Poisson distribution is later on
customers_daily <- rpois(number_of_stores, lambda = 50)

income_daily <- rnorm(number_of_stores, mean = 2000, sd = 100)

colors <- sample(c("green", "blue", "red", "white"),
                 size = number_of_stores, replace = TRUE)


df_inventado <- tibble(
  ind = indices,
  long = length_sim,
  ancho = width_sim,
  clientes = customers_daily,
  euros = income_daily,
  col = colors
)
```

```r
glimpse(df_inventado)
```

```
## Rows: 20
## Columns: 6
## $ ind      <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18~
## $ long     <dbl> 7.727768, 5.241214, 8.033535, 6.067075, 5.908894, 6.080340, 8~
## $ ancho    <dbl> 12.958071, 10.644176, 10.474705, 7.838526, 11.020973, 9.76414~
## $ clientes <int> 50, 38, 52, 42, 58, 47, 45, 39, 42, 49, 57, 59, 46, 57, 47, 4~
## $ euros    <dbl> 2016.820, 1979.281, 1771.956, 2158.435, 1951.960, 1988.361, 2~
## $ col      <chr> "white", "blue", "white", "blue", "blue", "green", "green", "~
```

Exercises. Now you make some calculations. You will need `mutate()`. Remember what has already been explained above. You can decide when to overwrite the data frame with the new calculations or just print them on the console.

1. Compute the total area of each store.
2. Calculate how many euros each customer spends, per store.
3. Calculate how many euros each customer spends on average, in total. *Hint.* For doing this, `summarise()` is very useful and you can read the documentation for learning how to use it. However, you can also use an approach with vectors using `$` and the functions `sum()` or `mean()`.
4. If the store is white or blue, reduce its length in 5 meters; else, increase it 10 meters. *Hint.* `? if_else()`. You should also write the condition with

`%in%`. For learning how to do this, play in the console with `"white" %in% c("white", "blue")` or `"red" %in% c("white", "blue")` and try to understand what's happening and how you can use it inside `mutate()`.

5. If all the customers in a day went to the store at the same time, how many squared meters per customer would there be in each store?

Solutions.

```
# Exercise 1
df_inventado %>%
  mutate(area = ancho * long)

# Exercise 2
df_inventado %>%
  mutate(euros / clientes)

# Exercise 3
df_inventado %>%
  summarise(sum(euros) / sum(clientes))

sum(df_inventado$euros) / sum(df_inventado$clientes)


# Exercise 4
df_inventado %>%
  mutate(nueva_long = if_else(col %in%  c("white", "blue"), long - 5, long + 10)) %>%
  select(nueva_long, long)

# Exercise 5
df_inventado %>%
  mutate(area = long * ancho, sqm_per_cust = area / clientes) %>%
  select(clientes, area, sqm_per_cust)
```

3.2 Filtering rows

You may know how to apply filters in Excel. In R you can do this too. The goal is to access the rows of a data frame that fulfill certain conditions. We'll work with an example.

If you have already run `library(dplyr)`, you will be able to use the `starwars` data frame. You can have a look a it with `glimpse()` (as you should know by now).

```
glimpse(starwars)
```

```
## Rows: 87
## Columns: 14
## $ name       <chr> "Luke Skywalker", "C-3PO", "R2-D2", "Darth Vader", "Leia Or~
## $ height     <int> 172, 167, 96, 202, 150, 178, 165, 97, 183, 182, 188, 180, 2~
## $ mass       <dbl> 77.0, 75.0, 32.0, 136.0, 49.0, 120.0, 75.0, 32.0, 84.0, 77.~
## $ hair_color <chr> "blond", NA, NA, "none", "brown", "brown, grey", "brown", N~
## $ skin_color <chr> "fair", "gold", "white, blue", "white", "light", "light", "~
## $ eye_color  <chr> "blue", "yellow", "red", "yellow", "brown", "blue", "blue",~
## $ birth_year <dbl> 19.0, 112.0, 33.0, 41.9, 19.0, 52.0, 47.0, NA, 24.0, 57.0, ~
## $ sex        <chr> "male", "none", "none", "male", "female", "male", "female",~
## $ gender     <chr> "masculine", "masculine", "masculine", "masculine", "femini~
## $ homeworld  <chr> "Tatooine", "Tatooine", "Naboo", "Tatooine", "Alderaan", "T~
## $ species    <chr> "Human", "Droid", "Droid", "Human", "Human", "Human", "Huma~
## $ films      <list> <"The Empire Strikes Back", "Revenge of the Sith", "Return~
## $ vehicles   <list> <"Snowspeeder", "Imperial Speeder Bike">, <>, <>, <>, "Imp~
## $ starships  <list> <"X-wing", "Imperial shuttle">, <>, <>, "TIE Advanced x1",~
```

Exercise.

- What are the names of the columns?
- What are the classes of each column?
- What are the dimensions of the data frame (number of rows and columns)?

Remark. Remember to use `? starwars` to access the documentation.

We can set a condition a take only the rows under this condition. For instance, we want all the character taller than 175cms. You may remember from vector that you can do create a logical vector with this condition doing something like this:

```
starwars$height > 175
```

```
##  [1] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
## [25] FALSE  TRUE FALSE    NA FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
## [37]  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE
## [49]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
## [61] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE
## [73] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE    NA    NA    NA
## [85]    NA    NA FALSE
```

Now we could see the names of the characters whose associated value in this vector is `TRUE`. The names of the characters are stored on the first column, `name`.

```
starwars$name[starwars$height > 175]
```

```
##  [1] "Darth Vader"          "Owen Lars"            "Biggs Darklighter"
##  [4] "Obi-Wan Kenobi"       "Anakin Skywalker"     "Wilhuff Tarkin"
##  [7] "Chewbacca"            "Han Solo"             "Jek Tono Porkins"
## [10] "Boba Fett"            "IG-88"                "Bossk"
## [13] "Lando Calrissian"     "Ackbar"               NA
## [16] "Qui-Gon Jinn"         "Nute Gunray"          "Jar Jar Binks"
## [19] "Roos Tarpals"         "Rugor Nass"           "Ric Olié"
## [22] "Quarsh Panaka"        "Bib Fortuna"          "Ayla Secura"
## [25] "Mace Windu"           "Ki-Adi-Mundi"         "Kit Fisto"
## [28] "Adi Gallia"           "Saesee Tiin"          "Yarael Poof"
## [31] "Plo Koon"             "Mas Amedda"           "Gregar Typho"
## [34] "Cliegg Lars"          "Poggle the Lesser"    "Dooku"
## [37] "Bail Prestor Organa" "Jango Fett"           "Dexter Jettster"
## [40] "Lama Su"              "Taun We"              "Wat Tambor"
## [43] "San Hill"             "Shaak Ti"             "Grievous"
## [46] "Tarfful"              "Raymus Antilles"      "Sly Moore"
## [49] "Tion Medon"           NA                     NA
## [52] NA                     NA                     NA
```

Yes, there are some `NA` things. We'll talk about than in a few minutes.

Working with vector is efficient but the sintax is verbose. dplyr provide us with a different sintax, data frame - orientated (remember that when doing statistics, data will be mainly stored in tables, therefore we love data frames and working with them <3).

Let's see how to do this with data frames.

```
starwars %>%
  filter(height > 175)
```

```
## # A tibble: 48 x 14
##    name    height  mass hair_color  skin_color eye_color birth_year sex   gender
##    <chr>    <int> <dbl> <chr>       <chr>      <chr>          <dbl> <chr> <chr>
##  1 Darth ~    202 136   none        white      yellow          41.9 male  mascu~
##  2 Owen L~    178 120   brown, grey light      blue            52   male  mascu~
##  3 Biggs ~    183  84   black       light      brown           24   male  mascu~
##  4 Obi-Wa~    182  77   auburn, wh~ fair       blue-gray       57   male  mascu~
##  5 Anakin~    188  84   blond       fair       blue            41.9 male  mascu~
##  6 Wilhuf~    180  NA   auburn, gr~ fair       blue            64   male  mascu~
```

```
##  7 Chewba~    228 112   brown       unknown    blue           200   male  mascu~
##  8 Han So~    180  80   brown       fair       brown           29   male  mascu~
##  9 Jek To~    180 110   brown       fair       blue            NA   male  mascu~
## 10 Boba F~    183  78.2 black       fair       brown           31.5 male  mascu~
## # ... with 38 more rows, and 5 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

Doing this we keep all the information about the characters: we have only removed the rows about characters we're not interested on because of their height.

We can also extend the pipeline and also select only the column of names.

```
starwars %>%
  filter(height > 175) %>%
  select(name)
```

```
## # A tibble: 48 x 1
##    name
##    <chr>
##  1 Darth Vader
##  2 Owen Lars
##  3 Biggs Darklighter
##  4 Obi-Wan Kenobi
##  5 Anakin Skywalker
##  6 Wilhuff Tarkin
##  7 Chewbacca
##  8 Han Solo
##  9 Jek Tono Porkins
## 10 Boba Fett
## # ... with 38 more rows
```

Remark. Note that we haven't used the `$` notation here. When working with dplyr, R knows where to look for the columns: it knows they are in the data frame. Thus, there is no need to write the name of the data frame followed by `$` for accessing columns. Careful with this: it is not a mistake doing it, but works differently and it can lead to errors in some situations. Best practice: don't do it. Never.

For numerical columns you can use all the comparisons you know: `>`, `>=`, `<`, `<=`, `==` and `!=` (for distinct). Notice that you cannot use one single equal sign for comparing. It is very common this mistake but dplyr will properly warn you: just read the messages.

Another example:

```
starwars %>%
  filter(height != 202) %>%
  nrow()
```

```
## [1] 80
```

You can work with character columns:

```
starwars %>%
  filter(hair_color == "brown")
```

```
## # A tibble: 18 x 14
##    name     height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 Leia Or~    150    49 brown      light      brown             19 fema~ femin~
##  2 Beru Wh~    165    75 brown      light      blue              47 fema~ femin~
##  3 Chewbac~    228   112 brown      unknown    blue             200 male  mascu~
##  4 Han Solo    180    80 brown      fair       brown             29 male  mascu~
##  5 Wedge A~    170    77 brown      fair       hazel             21 male  mascu~
##  6 Jek Ton~    180   110 brown      fair       blue              NA male  mascu~
##  7 Arvel C~     NA    NA brown      fair       brown             NA male  mascu~
##  8 Wicket ~     88    20 brown      brown      brown              8 male  mascu~
##  9 Qui-Gon~    193    89 brown      fair       blue              92 male  mascu~
## 10 Ric Olié    183    NA brown      fair       blue              NA <NA>  <NA>
## 11 Cordé       157    NA brown      light      brown             NA fema~ femin~
## 12 Cliegg ~    183    NA brown      fair       blue              82 male  mascu~
## 13 Dormé       165    NA brown      light      brown             NA fema~ femin~
## 14 Tarfful     234   136 brown      brown      blue              NA male  mascu~
## 15 Raymus ~    188    79 brown      light      brown             NA male  mascu~
## 16 Rey          NA    NA brown      light      hazel             NA fema~ femin~
## 17 Poe Dam~     NA    NA brown      light      brown             NA male  mascu~
## 18 Padmé A~    165    45 brown      light      brown             46 fema~ femin~
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

And you can combine rules. With commas if you want to have all the rules at the same time: *'people whose skin color is light and height is greater or equal than 165:'*

```
starwars %>%
  filter(skin_color == "light", height >= 165)
```

```
## # A tibble: 7 x 14
```

```
##   name     height  mass hair_color  skin_color eye_color birth_year sex   gender
##   <chr>     <int> <dbl> <chr>       <chr>      <chr>          <dbl> <chr> <chr>
## 1 Owen La~    178   120 brown, grey light      blue              52 male  mascu~
## 2 Beru Wh~    165    75 brown       light      blue              47 fema~ femin~
## 3 Biggs D~    183    84 black       light      brown             24 male  mascu~
## 4 Lobot       175    79 none        light      blue              37 male  mascu~
## 5 Dormé       165    NA brown       light      brown             NA fema~ femin~
## 6 Raymus ~    188    79 brown       light      brown             NA male  mascu~
## 7 Padmé A~    165    45 brown       light      brown             46 fema~ femin~
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

If you want say *'or'* you need to use the | symbol: *'people whose skin color is light or whose height is lower than 100:'*

```
starwars %>%
  filter(skin_color == "light" | height < 100)
```

```
## # A tibble: 18 x 14
##    name     height  mass hair_color skin_color eye_color birth_year sex   gender
##    <chr>     <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
##  1 R2-D2        96    32 <NA>       white, bl~ red               33 none  mascu~
##  2 Leia Or~    150    49 brown      light      brown             19 fema~ femin~
##  3 Owen La~    178   120 brown, gr~ light      blue              52 male  mascu~
##  4 Beru Wh~    165    75 brown      light      blue              47 fema~ femin~
##  5 R5-D4        97    32 <NA>       white, red red               NA none  mascu~
##  6 Biggs D~    183    84 black      light      brown             24 male  mascu~
##  7 Yoda         66    17 white      green      brown            896 male  mascu~
##  8 Lobot       175    79 none       light      blue              37 male  mascu~
##  9 Wicket ~     88    20 brown      brown      brown              8 male  mascu~
## 10 Dud Bolt     94    45 none       blue, grey yellow            NA male  mascu~
## 11 Cordé       157    NA brown      light      brown             NA fema~ femin~
## 12 Dormé       165    NA brown      light      brown             NA fema~ femin~
## 13 Ratts T~     79    15 none       grey, blue unknown           NA male  mascu~
## 14 R4-P17       96    NA none       silver, r~ red, blue         NA none  femin~
## 15 Raymus ~    188    79 brown      light      brown             NA male  mascu~
## 16 Rey          NA    NA brown      light      hazel             NA fema~ femin~
## 17 Poe Dam~     NA    NA brown      light      brown             NA male  mascu~
## 18 Padmé A~    165    45 brown      light      brown             46 fema~ femin~
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

3.3 Distinct

In Excel you can easily get the different values from a column when selecting it. Imaging that you want all the hair colors available in the Starwars universe, no matter whose hair it is. In this case, you would be interested on the distinct values of the `hair_color` of the data frame.

There are two ways of doing this: the vectorial way and the dplyr way. Depending on the situation you will need one or the other, so learn both.

3.3.1 R base

Exercise. Get the unique values for the `hair_color` columns. Remember to use the `$` notation for getting the vector and then apply the function `unique()` to it. You will have a vector with no repeated values.

```
unique(starwars$hair_color)
```

```
##  [1] "blond"         NA              "none"          "brown"
##  [5] "brown, grey"   "black"         "auburn, white" "auburn, grey"
##  [9] "white"         "grey"          "auburn"        "blonde"
## [13] "unknown"
```

Don't worry too much about that `NA` thing: we'll get into that in the next section.

3.3.2 dplyr

As you have seen, applying `unique()` to a vector you obtain a vector of the different values of the input. But when dealing with exploratory analysis you may want to preserve the tabular format, since it is easy to glance through it as well as export to Excel. For doing this, we use the dplyr function `distinct()`, which receives a data frame or tibble via the pipe `%>%` and the name of the columns whose unique values we want to obtain within the brackets.

```
starwars %>%
  distinct(hair_color)
```

```
## # A tibble: 13 x 1
##    hair_color
##    <chr>
##  1 blond
##  2 <NA>
```

```
##  3 none
##  4 brown
##  5 brown, grey
##  6 black
##  7 auburn, white
##  8 auburn, grey
##  9 white
## 10 grey
## 11 auburn
## 12 blonde
## 13 unknown
```

We can do the same with several columns at the same time:

```
starwars %>%
  distinct(hair_color, skin_color)
```

```
## # A tibble: 50 x 2
##    hair_color    skin_color
##    <chr>         <chr>
##  1 blond         fair
##  2 <NA>          gold
##  3 <NA>          white, blue
##  4 none          white
##  5 brown         light
##  6 brown, grey   light
##  7 <NA>          white, red
##  8 black         light
##  9 auburn, white fair
## 10 auburn, grey  fair
## # ... with 40 more rows
```

In this data frame you have all the different existing combinations between `hair_color` and `skin_color`. There is no way of doing this only with `unique()`, in a vectorial way.

Exercise (also for readr). Get the distinct combinations between `eye_color` and `gender` and create a data frame with them. Export this data frame or tibble to a csv file using a function from the readr library. *Hint.* After running `library(readr)`, write on the console or the script `write_` and let the autocompletion propose you some alternatives. Decide which function you should use. *Another hint.* There are several solutions for exporting the file. Anyway, remember to use `?` for reading the help of a function.

```
new_df <- starwars %>%
  distinct(eye_color, gender)

library(readr)
write_csv(new_df, "new_file_super_cool.csv")
```

3.4 Some comments about NA

When having a look at the values of `hair_color` you can see there is something written like `NA`. This is a symbol used for specifying the generally named missing values and its direct meaning is *not available*. In several programming languages, like R, this symbol is very important because R works with it in a particular way.

Missing values cannot easily be replace by some other because it will often have meaning. We will mention some examples during the sessions.

Let's extract the distinct values of that column.

```
unique_hair_color <- unique(starwars$hair_color)
unique_hair_color
```

```
##  [1] "blond"         NA              "none"          "brown"
##  [5] "brown, grey"   "black"         "auburn, white" "auburn, grey"
##  [9] "white"         "grey"          "auburn"        "blonde"
## [13] "unknown"
```

This `NA` is the only value not written within `"`. But the vector is a character vector.

```
class(unique_hair_color)
```

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

If take the first value of the vector, `"blonde"`, it is a character:

```
class(unique_hair_color[1])
```

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

What about the second, which is `NA`?

```
class(unique_hair_color[2])
```

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

Exercise. Repeat the same procedure with `height` column and see what class `NA` belong to this time.

Exercise. Apply the `class()` function to `NA` and see the result.

This is very important. `NA` class is not fixed and can lead you to a lot of problems when reading data.

It is mandatory knowing how to deal with `NA` values since it can change everything you analyse. For instance, let's calculate the average height of the characters from Starwars.

```
mean(starwars$height)
```

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

The result of that calculation is `NA` because when operating with `NA` the result is always `NA`, even the most simple thing:

```
1 + NA
```

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

You cannot operate with `NA` so you need to get rid of it. Some R functions allow you to exclude them adding some commands:

```
mean(starwars$height, na.rm = TRUE)
```

```
## [1] 174.358
```

We will now focus on dealing with them in dplyr.

`is.na()` is a R base function that allows us detecting `NA` values. Simple example:

```
vector_with_NA <- c(1, NA, 3)
is.na(vector_with_NA)
```

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

You can also work denying it, in logical terms:

```
!is.na(vector_with_NA)
```

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

Working with dplyr is similar:

```
starwars %>%
  filter(is.na(height))
```

```
## # A tibble: 6 x 14
##   name     height  mass hair_color skin_color eye_color birth_year sex    gender
##   <chr>     <int> <dbl> <chr>      <chr>      <chr>           <dbl> <chr>  <chr>
## 1 Arvel C~     NA    NA brown      fair       brown              NA male   mascu~
## 2 Finn         NA    NA black      dark       dark               NA male   mascu~
## 3 Rey          NA    NA brown      light      hazel              NA female femin~
## 4 Poe Dam~     NA    NA brown      light      brown              NA male   mascu~
## 5 BB8          NA    NA none       none       black              NA none   mascu~
## 6 Captain~     NA    NA unknown    unknown    unknown            NA <NA>   <NA>
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

The contrary:

```
starwars %>%
  filter(!is.na(height))
```

```
## # A tibble: 81 x 14
##    name    height  mass hair_color  skin_color eye_color birth_year sex   gender
##    <chr>    <int> <dbl> <chr>       <chr>      <chr>          <dbl> <chr> <chr>
##  1 Luke S~    172    77 blond       fair       blue            19   male  mascu~
##  2 C-3PO      167    75 <NA>        gold       yellow         112   none  mascu~
##  3 R2-D2       96    32 <NA>        white, bl~ red             33   none  mascu~
##  4 Darth ~    202   136 none        white      yellow          41.9 male  mascu~
##  5 Leia O~    150    49 brown       light      brown           19   fema~ femin~
##  6 Owen L~    178   120 brown, grey light      blue            52   male  mascu~
##  7 Beru W~    165    75 brown       light      blue            47   fema~ femin~
##  8 R5-D4       97    32 <NA>        white, red red             NA   none  mascu~
##  9 Biggs ~    183    84 black       light      brown           24   male  mascu~
## 10 Obi-Wa~    182    77 auburn, wh~ fair       blue-gray       57   male  mascu~
## # ... with 71 more rows, and 5 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

It is possible to combine rules, as we previously saw.

```
starwars %>%
  filter(is.na(height) | is.na(hair_color))
```

```
## # A tibble: 11 x 14
##    name    height  mass hair_color skin_color  eye_color birth_year sex   gender
##    <chr>    <int> <dbl> <chr>      <chr>       <chr>           <dbl> <chr> <chr>
##  1 C-3PO      167    75 <NA>       gold        yellow            112 none  mascu~
##  2 R2-D2       96    32 <NA>       white, blue red                33 none  mascu~
##  3 R5-D4       97    32 <NA>       white, red  red                NA none  mascu~
##  4 Greedo     173    74 <NA>       green       black              44 male  mascu~
##  5 Jabba ~    175  1358 <NA>       green-tan,~ orange            600 herm~ mascu~
##  6 Arvel ~     NA    NA brown      fair        brown              NA male  mascu~
##  7 Finn        NA    NA black      dark        dark               NA male  mascu~
##  8 Rey         NA    NA brown      light       hazel              NA fema~ femin~
##  9 Poe Da~     NA    NA brown      light       brown              NA male  mascu~
## 10 BB8         NA    NA none       none        black              NA none  mascu~
## 11 Captai~     NA    NA unknown    unknown     unknown            NA <NA>  <NA>
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

This is useful for making sure that you have all the information you need for an analysis. Suppose you are asked to calculate the BMI (explained during the second session). You need both the height and the mass, so you should check both columns are available.

Exercise. Keep only the rows with both height and mass available and create a new column with the BMI (m/h^2). Finally select the name of the character and the new column.

```
starwars %>%
  filter(!is.na(height), !is.na(mass)) %>%
  mutate(mass / (height ^ 2))
```

```
## # A tibble: 59 x 15
##    name    height  mass hair_color  skin_color eye_color birth_year sex   gender
##    <chr>    <int> <dbl> <chr>       <chr>      <chr>           <dbl> <chr> <chr>
##  1 Luke S~    172    77 blond       fair       blue             19   male  mascu~
##  2 C-3PO      167    75 <NA>        gold       yellow          112   none  mascu~
##  3 R2-D2       96    32 <NA>        white, bl~ red              33   none  mascu~
##  4 Darth ~    202   136 none        white      yellow           41.9 male  mascu~
##  5 Leia O~    150    49 brown       light      brown            19   fema~ femin~
##  6 Owen L~    178   120 brown, grey light      blue             52   male  mascu~
##  7 Beru W~    165    75 brown       light      blue             47   fema~ femin~
##  8 R5-D4       97    32 <NA>        white, red red              NA   none  mascu~
```

```
##  9 Biggs ~    183    84 black       light      brown           24   male  mascu~
## 10 Obi-Wa~    182    77 auburn, wh~ fair       blue-gray       57   male  mascu~
## # ... with 49 more rows, and 6 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>, mass/(height^2) <dbl>
```

Exercise. Read the help of `na.omit()` function for removing all the rows with at least one `NA` value from the `starwars` data frame. Create a new dataframe with the data. Export the new data frame to a csv file (separated with **comma**).

3.5 Row numbers

Another way for selecting rows is using a numerical index instead of conditions. Thus, you could access directly the first, third and fifth column of a data frame. This can be done with `slice()` (also with R base, but we're not getting into that).

```
iris %>%
  slice(1:3, 100:103)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1          5.1         3.5          1.4         0.2     setosa
## 2          4.9         3.0          1.4         0.2     setosa
## 3          4.7         3.2          1.3         0.2     setosa
## 4          5.7         2.8          4.1         1.3 versicolor
## 5          6.3         3.3          6.0         2.5  virginica
## 6          5.8         2.7          5.1         1.9  virginica
## 7          7.1         3.0          5.9         2.1  virginica
```

3.6 Exercises

1. Read the file `volpre2019.csv` and create a data frame with its data. Name it however you like. It stores data about the volume and the price of lots of products at MercaMadrid.
2. Call the library janitor (install it if needed) and use the `clean_names()` function on the data frame (overwrite it).
3. Explore the data frame with the functions you know. `nrow`, `ncol` (you can use `dim()` instead), `glimpse()`. Remember to use `summary()` too.
4. Count how many `NA` values there are in the `fecha_desde` column. *Hint.* For now it is OK if you just use `is.na()` for building a logical vector and then `sum()` for adding the number of cases with `NA`.

5. Exclude the cases with `fecha_desde` as `NA` and overwrite the data frame.
6. Get the distinct origins (`desc_origin` column) of `"VACUNO"` productos (`desc_variedad_2`).
7. Select four products from the `desc_variedad_2` column and extract the months when they are available (`fecha_desde`) and the origin. Do it separately for each of them. The final data frame for each product should have two columns. Arrange that data frame by `desc_origin`. The function you need is `arrange()`. *Suggestion.* For selecting the products, I used `distinct()` on the column and then `sample_n(4)`, everything linked with pipes `%>%`. Read the documentation on `sample_n()` if needed.

Solutions.

```
# Exercise 1
library(readr)
df_merca <- read_csv2("data/volpre2019.csv")
```

```
## i Using "','" as decimal and "'.'" as grouping mark. Use `read_delim()` for more co
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   `Fecha Desde` = col_double(),
##   `Fecha Hasta` = col_double(),
##   `Cod variedad` = col_character(),
##   `Desc variedad` = col_character(),
##   `Cod Origen` = col_double(),
##   `Desc Origen` = col_character(),
##   Kilos = col_double(),
##   `Precio frecuente` = col_double(),
##   `Precio maximo` = col_double(),
##   `Precio minimo` = col_double(),
##   `Desc Variedad 2` = col_character()
## )
```

```
# Exercise 2
library(janitor)
```

```
##
## Attaching package: 'janitor'
```

```
## The following objects are masked from 'package:stats':
##
##     chisq.test, fisher.test
```

```
df_merca <- clean_names(df_merca)

# Exercise 4
sum(is.na(df_merca$fecha_desde))
```

```
## [1] 5
```

```
# Exercise 5
df_merca <- df_merca %>%
  filter(!is.na(fecha_desde))

# Exercise 6
df_merca %>%
  filter(desc_variedad_2 == "VACUNO") %>%
  distinct(desc_origen)
```

```
## # A tibble: 59 x 1
##    desc_origen
##    <chr>
##  1 BADAJOZ
##  2 BARCELONA
##  3 CACERES
##  4 CUENCA
##  5 LUGO
##  6 MADRID
##  7 NAVARRA
##  8 ORENSE
##  9 PONTEVEDRA
## 10 SALAMANCA
## # ... with 49 more rows
```

```
# Exercise 7
df_merca %>%
  distinct(desc_variedad_2) %>%
  sample_n(4)
```

```
## # A tibble: 4 x 1
##   desc_variedad_2
##   <chr>
## 1 JUREL
## 2 PRECOCINADOS
## 3 BERROS
## 4 POMELO
```

```
df_merca %>%
  filter(desc_variedad_2 == "CONCHA") %>%
  select(fecha_desde, desc_origen) %>%
  arrange(desc_origen)
```

```
## # A tibble: 17 x 2
##    fecha_desde desc_origen
##          <dbl> <chr>
##  1    20190201 CADIZ
##  2    20190501 CADIZ
##  3    20190601 CADIZ
##  4    20190101 FRANCIA
##  5    20190601 GRAN BRETAÑA
##  6    20190501 GUIPUZCOA
##  7    20190601 GUIPUZCOA
##  8    20190101 HUELVA
##  9    20190501 HUELVA
## 10    20190301 LA CORUÑA
## 11    20190501 LA CORUÑA
## 12    20190101 PONTEVEDRA
## 13    20190201 PONTEVEDRA
## 14    20190301 PONTEVEDRA
## 15    20190401 PONTEVEDRA
## 16    20190501 PONTEVEDRA
## 17    20190601 PONTEVEDRA
```

3.7 Summarizing tables

The main library we'll use is dplyr.

```
library(dplyr)
```

We will be working with the storm dataset, stored on the `storms.txt` file.

```
library(readr)
df_storms <- read_tsv("data/storms.txt",
                      col_types = cols(
                        name = col_character(),
                        year = col_double(),
                        month = col_double(),
                        day = col_double(),
                        hour = col_double(),
```

```
                        status = col_character(),
                        category = col_factor(),
                        wind = col_integer(),
                        pressure = col_integer(),
                        ts_diameter = col_double(),
                        hu_diameter = col_double()
                      ))

glimpse(df_storms)
```

```
## Rows: 10,010
## Columns: 11
## $ name        <chr> "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "A~
## $ year        <dbl> 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975~
## $ month       <dbl> 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7~
## $ day         <dbl> 27, 27, 27, 27, 28, 28, 28, 28, 29, 29, 29, 29, 30, 30, 30~
## $ hour        <dbl> 0, 6, 12, 18, 0, 6, 12, 18, 0, 6, 12, 18, 0, 6, 12, 18, 0,~
## $ status      <chr> "tropical depression", "tropical depression", "tropical de~
## $ category    <fct> -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ wind        <int> 25, 25, 25, 25, 25, 25, 25, 30, 35, 40, 45, 50, 50, 55, 60~
## $ pressure    <int> 1013, 1013, 1013, 1013, 1012, 1012, 1011, 1006, 1004, 1002~
## $ ts_diameter <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ hu_diameter <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
```

3.7.1 Exercises

1. What can you learn applying `summary()` to the tibble?
2. Which are the different values at the `name` column? These are the storms and hurricanes we wil be working with. Create a tibble with one column containing this unique names.
3. Which are the different available status? Don't create the tibble: just print it on the console.
4. Which are the different combinations between status and pressure?

3.8 Aggregating the data

Exercise 1. How would you calculate the average wind for each status?

```
## [1] "Tropical depression: 27.2691552062868"
```

```
## [1] "Tropical storm: 45.8058984910837"
```

```
## [1] "Hurricane: 85.9689420899385"
```

Taking into account only what has been studied so far about dplyr, we would need to jump into vectors again to calculate the mean for these cases. However we have another function in dplyr for simplifying this: `summarise()`.

```
df_storms %>%
  filter(status == "tropical depression") %>%
  summarise(avg_wind = mean(wind))
```

```
## # A tibble: 1 x 1
##   avg_wind
##      <dbl>
## 1     27.3
```

```
df_storms %>%
  filter(status == "tropical storm") %>%
  summarise(avg_wind = mean(wind))
```

```
## # A tibble: 1 x 1
##   avg_wind
##      <dbl>
## 1     45.8
```

```
df_storms %>%
  filter(status == "hurricane") %>%
  summarise(avg_wind = mean(wind))
```

```
## # A tibble: 1 x 1
##   avg_wind
##      <dbl>
## 1     86.0
```

You can calculate several columns on the fly.

```
df_storms %>%
  filter(status == "hurricane") %>%
  summarise(avg_wind = mean(wind),
            sd_wind = sd(wind))
```

```
## # A tibble: 1 x 2
##   avg_wind sd_wind
##      <dbl>   <dbl>
## 1     86.0    20.3
```

```
glimpse(df_storms)
```

```
## Rows: 10,010
## Columns: 11
## $ name        <chr> "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "A~
## $ year        <dbl> 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975~
## $ month       <dbl> 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7~
## $ day         <dbl> 27, 27, 27, 27, 28, 28, 28, 28, 29, 29, 29, 29, 30, 30, 30~
## $ hour        <dbl> 0, 6, 12, 18, 0, 6, 12, 18, 0, 6, 12, 18, 0, 6, 12, 18, 0,~
## $ status      <chr> "tropical depression", "tropical depression", "tropical de~
## $ category    <fct> -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ wind        <int> 25, 25, 25, 25, 25, 25, 25, 30, 35, 40, 45, 50, 50, 55, 60~
## $ pressure    <int> 1013, 1013, 1013, 1013, 1012, 1012, 1011, 1006, 1004, 1002~
## $ ts_diameter <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ hu_diameter <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
```

Exercise 2. For the storms, depressions and hurricanes that took place between 1975 and 1980, create a new column with the mean and standard deviation of the wind. Create a new column substacting the mean from wind and then divide by the standard deviation. Calculate with `summarise()` the mean and the standard deviation of this new column. What happened?

```
## # A tibble: 1 x 2
##   `mean(new_column)` `sd(new_column)`
##                <dbl>            <dbl>
## 1          -2.07e-17                1
```

Exercise 3.

1. For the registers where `hu_diameter` is not `NA`, calculate the minimun pressure, the median, the average and the maximum

```
## # A tibble: 1 x 4
##     min media mediana   max
##   <int> <dbl>   <dbl> <int>
## 1   882  991.     998  1017
```

2. Are there more registers below the average or above?

```
## # A tibble: 1 x 2
##   above_avg below_avg
##       <int>     <int>
## 1      2254      1228
```

Exercise 4. For the hurricane, calculate the average `ts_diameter`. *Hint.* It is not `NA`.

```
## # A tibble: 1 x 1
##   media
##   <dbl>
## 1  288.
```

3.9 A summary

We are now working with an example from the official dplyr documentation, available on the tidyverse site. This notes are just a summary of that page, consisting mainly in copy-pasted paragraphs. The original website may be too technical for our purposes.

The data will used is a dataset containing all 336776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics, it is available in the `nycflights13` package and is documented in `?nycflights13`.

```
library(dplyr)
library(nycflights13)

dim(flights)
```

```
## [1] 336776     19
```

```
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

3.10 Dplyr verbs

Dplyr aims to provide a function for each basic verb of data manipulation:

- `filter()` to select cases based on their values.
- `arrange()` to reorder the cases.
- `select()` and `rename()` to select variables based on their names.
- `mutate()` and `transmute()` to add new variables that are functions of existing variables.
- `summarise()` to condense multiple values to a single value.
- `sample_n()` and `sample_frac()` to take random samples.

3.10.1 Filter rows with filter()

`filter()` allows you to select a subset of rows in a data frame. Firstly, the function receives a tibble (we are doing during the sessions via the `%>%` command but it is not necessary). Then we also have to write inside the brackets the expression used for rows selection. This expression will be `TRUE` for the selected rows.

Exercise 1. Select all flights on January 1st.

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 832 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

Exercise 2. How many flights departed from JFK airport?

```
## [1] 111279
```

Exercise 3. How many delayed flights were there?

```
## [1] 133004
```

Exercise 4. Select the flights with no NA data on the `dep_delay` column.

```
## # A tibble: 328,521 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 328,511 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

3.10.2 Arrange rows with `arrange()`

`arrange()` reorders the rows. It takes a data frame, and a set of column names to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.

```
flights %>% arrange(year, month, day)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

```
flights %>% arrange(desc(arr_delay))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     9      641            900      1301     1242           1530
##  2  2013     6    15     1432           1935      1137     1607           2120
##  3  2013     1    10     1121           1635      1126     1239           1810
##  4  2013     9    20     1139           1845      1014     1457           2210
##  5  2013     7    22      845           1600      1005     1044           1815
##  6  2013     4    10     1100           1900       960     1342           2211
##  7  2013     3    17     2321            810       911      135           1020
##  8  2013     7    22     2257            759       898      121           1026
##  9  2013    12     5      756           1700       896     1058           2020
## 10  2013     5     3     1133           2055       878     1250           2215
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

3.10.3 Select columns with `select()`

Often you work with large datasets with many columns but only a few are actually of interest to you. `select()` allows you to rapidly zoom in on a useful subset, even using operations.

```
flights %>% select(year, month, day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

```
flights %>% select(year:day) # Everything between year and day
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

```
flights %>% select(-(year:day)) # Everything except year, day and what is in the middl
```

```
## # A tibble: 336,776 x 16
##    dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
##       <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
##  1      517            515         2      830            819        11 UA
##  2      533            529         4      850            830        20 UA
##  3      542            540         2      923            850        33 AA
##  4      544            545        -1     1004           1022       -18 B6
##  5      554            600        -6      812            837       -25 DL
##  6      554            558        -4      740            728        12 UA
##  7      555            600        -5      913            854        19 B6
##  8      557            600        -3      709            723       -14 EV
##  9      557            600        -3      838            846        -8 B6
## 10      558            600        -2      753            745         8 AA
## # ... with 336,766 more rows, and 9 more variables: flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

There are a number of helper functions you can use within `select()`, like `starts_with()`, `ends_with()` and `contains()`. These let you quickly match larger blocks of variables that meet some criterion. See ?select for more details.

```
iris %>%
  select(starts_with("Petal")) %>%
  head(6)  # Used for limiting the number of printed rows
```

```
##   Petal.Length Petal.Width
## 1          1.4         0.2
## 2          1.4         0.2
## 3          1.3         0.2
## 4          1.5         0.2
## 5          1.4         0.2
## 6          1.7         0.4
```

Exercise 5. From `flights` tibble, select all the columns related to delays (containing something about delays in the name).

```
## # A tibble: 336,776 x 2
##    dep_delay arr_delay
##        <dbl>     <dbl>
##  1         2        11
##  2         4        20
##  3         2        33
##  4        -1       -18
##  5        -6       -25
##  6        -4        12
##  7        -5        19
##  8        -3       -14
##  9        -3        -8
## 10        -2         8
## # ... with 336,766 more rows
```

You can rename columns while selecting:

```
flights %>% select(tail_num = tailnum)
```

```
## # A tibble: 336,776 x 1
##    tail_num
##    <chr>
##  1 N14228
##  2 N24211
##  3 N619AA
##  4 N804JB
##  5 N668DN
##  6 N39463
##  7 N516JB
##  8 N829AS
##  9 N593JB
## 10 N3ALAA
## # ... with 336,766 more rows
```

But this is not useful when you want to keep all the variables. For just renaming we use `rename()`.

```
flights %>% names()
```

```
##  [1] "year"           "month"          "day"            "dep_time"
##  [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
##  [9] "arr_delay"      "carrier"        "flight"         "tailnum"
## [13] "origin"         "dest"           "air_time"       "distance"
## [17] "hour"           "minute"         "time_hour"
```

```
flights %>%
  rename(tail_num = tailnum) %>%
  names()
```

```
##  [1] "year"           "month"          "day"            "dep_time"
##  [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
##  [9] "arr_delay"      "carrier"        "flight"         "tail_num"
## [13] "origin"         "dest"           "air_time"       "distance"
## [17] "hour"           "minute"         "time_hour"
```

3.10.4 Add new columns with `mutate()`

The job of `mutate()` is redefining existing columns or adding new ones.

Exercise 6.

- Define a new column called gain, consisting on substracting `dep_delay` to `arr_delay`.
- Define another one as the average speed (`distance` divided by `air_time`). Multiply it by 60 to have the result in minutes.

```
## # A tibble: 336,776 x 21
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
```

```
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 13 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
## #   gain <dbl>, speed <dbl>
```

`mutate()` allows you to refer to columns that you've just created:

```
flights %>%
  mutate(
    gain = arr_delay - dep_delay,
    gain_per_hour = gain / (air_time / 60)
)
```

```
## # A tibble: 336,776 x 21
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ... with 336,766 more rows, and 13 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>,
## #   gain <dbl>, gain_per_hour <dbl>
```

3.10.5 Summarise values with `summarise()`

The last verb is `summarise()`. It collapses a data frame to a single row.

Exercise 7. Calculate the mean of the delay in the departure. Mind the NAs.

```
## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1  12.6
```

This is very powerful when combined with `group_by()`, not seen yet.

3.10.6 Randomly sample rows with `sample_n()` and `sample_frac()`

You can use `sample_n()` and `sample_frac()` to take a random sample of rows: use `sample_n()` for a fixed number and `sample_frac()` for a fixed fraction.

```
flights %>% sample_n(10)
```

```
## # A tibble: 10 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013    11    21      616            610         6      918            915
##  2  2013    11    26     1942           1830        72     2134           2046
##  3  2013     5    25      951            959        -8     1142           1203
##  4  2013    10    15     1600           1559         1     1859           1943
##  5  2013     2    27      554            600        -6      835            837
##  6  2013     9    26     1709           1710        -1     1827           1835
##  7  2013     9    17      835            840        -5      950           1022
##  8  2013     3    16     2056           2040        16     2356           2344
##  9  2013    11    10      755            746         9     1052           1050
## 10  2013    10    14     1049           1059       -10     1236           1254
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

```
flights %>% sample_frac(0.01)
```

```
## # A tibble: 3,368 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     5      950            904        46     1318           1210
##  2  2013     5    30     1741           1732         9     2148           2114
##  3  2013     1    22     1458           1459        -1     1723           1656
##  4  2013    10     4     1245           1249        -4     1459           1512
##  5  2013     2    15      603            600         3      913            906
##  6  2013     2    13      800            810       -10     1009           1007
##  7  2013     7    10      759            800        -1     1008           1021
##  8  2013    10     7     1822           1715        67     2021           1937
##  9  2013     3    28      724            730        -6     1028           1100
## 10  2013     2    13      843            840         3     1154           1143
## # ... with 3,358 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

3.10.7 Commonalities

Notice that all the functions return data frames. Therefore, you can link them all with `%>%`.

```
flights %>%
  select(arr_delay, dep_delay) %>%
  filter(arr_delay > 30 | dep_delay > 30) %>%
  summarise(
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  )
```

```
## # A tibble: 1 x 2
##     arr   dep
##   <dbl> <dbl>
## 1  76.3  72.1
```

3.11 Grouped operations

The dplyr verbs are useful on their own, but they become even more powerful when you apply them to groups of observations within a dataset. In dplyr, you do this with the `group_by()` function. It breaks down a dataset into specified groups of rows. When you then apply the verbs above on the resulting object they'll be automatically applied "by group".

Grouping affects the verbs as follows:

- grouped `select()` is the same as ungrouped `select()`, except that grouping variables are always retained.
- grouped `arrange()` is the same as ungrouped; unless you set `.by_group = TRUE`, in which case it orders first by the grouping variables
- `mutate()` and `filter()` are most useful in conjunction with window functions (like `rank()`, or `min(x) == x`). They are described in detail in `vignette("window-functions")`.
- `sample_n()` and `sample_frac()` sample the specified number/fraction of rows in each group.
- `summarise()` computes the summary for each group.

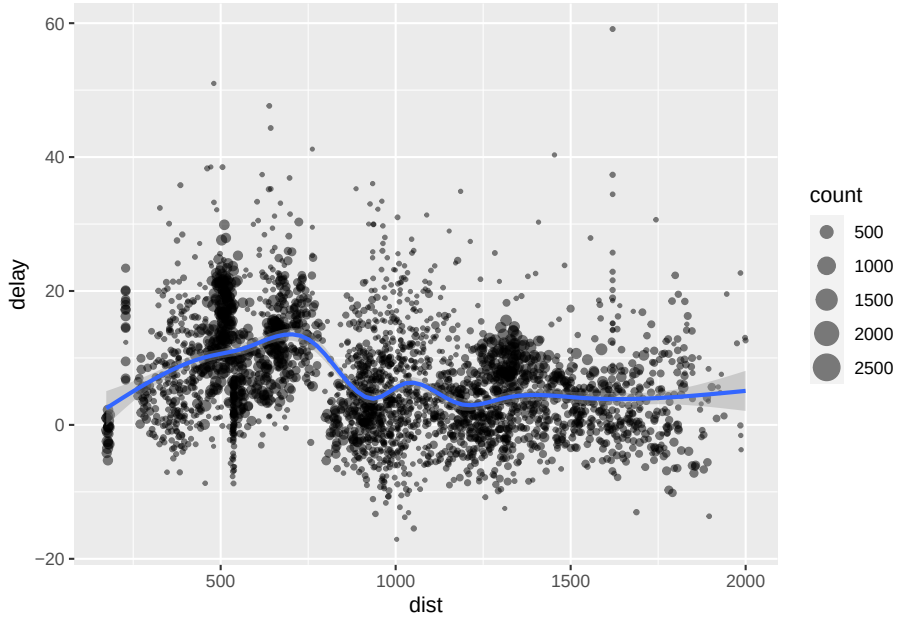
Exercise 8. Split the complete dataset into individual planes (`tailnum`) and then summarise each plane by counting the number of flights (`count = n()`) and computing the average distance (`distance`) and arrival delay (`arr_delay`). Create a tibble with that result and then use the provided ggplot code to make a plot with the distribution.

```
library(ggplot2)
ggplot(delay, aes(dist, delay)) +
  geom_point(aes(size = count), alpha = 1/2) +
  geom_smooth() +
  scale_size_area()
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

```
## Warning: Removed 1 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 1 rows containing missing values (geom_point).
```



You use `summarise()` with aggregate functions, which take a vector of values and return a single number. There are many useful examples of such functions in base R like `min()`, `max()`, `mean()`, `sum()`, `sd()`, `median()`, and `IQR()`. dplyr provides a handful of others:

- `n()`: the number of observations in the current group
- `n_distinct(x)`:the number of unique values in x.
- `first(x)`, `last(x)` and `nth(x, n)` - these work similarly to `x[1]`, `x[length(x)]`, and `x[n]` but give you more control over the result if the value is missing.

Exercise 9. Find the number of planes and the number of flights that go to each possible destination (`dest`).

```
## # A tibble: 105 x 3
##    dest  planes flights
##    <chr>  <int>   <int>
##  1 ABQ      108     254
##  2 ACK       58     265
##  3 ALB      172     439
##  4 ANC        6       8
##  5 ATL     1180   17215
##  6 AUS      993    2439
##  7 AVL      159     275
##  8 BDL      186     443
##  9 BGR       46     375
## 10 BHM       45     297
## # ... with 95 more rows
```

> There is more information about dplyr on the website but it may be too advanced for what we need in the course.

Chapter 4

Data visualization

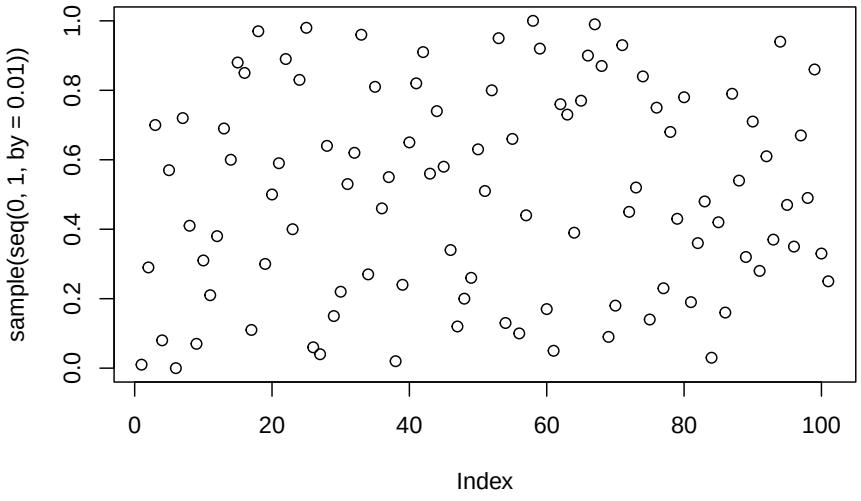
4.1 First steps into dataviz

When dealing with a very small set of data, a table can be enough for understanding it. But increasing the amount of data, even just a few rows, a proper plot can really help getting some information. Visualization is key in explaining data.

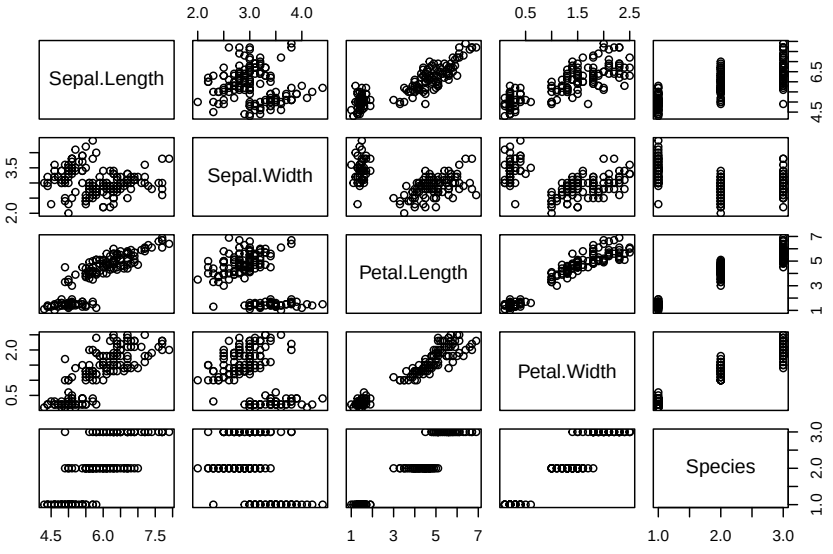
There exist several tools for visualizing data. Every stats software will provide you some packages for visualizing. There are even tools especially prepared for them, such as Qlik or PowerBI (the most typical aren't open software).

With R base we can make lots of plots, mainly via the `plot()` function. It is important knowing this functions, since it interacts differently with the object it receives, depending on its class. For instance:

```
plot(sample(seq(0, 1, by = 0.01)))
```




```
plot(iris)
```



However, we are going to work with ggplot2, the most common library for visualization in R. R base is usually said to be ugly for data visualization: this

is too subjective. What's true is the potential of ggplot2 is incredibly higher than base's one.

4.1.1 ggplot2

The *gg* in ggplot2 stands for *grammar of graphics*. ggplot2 provides us with a sintaxis for developing plots based on adding layers to a plot. A plot in ggplot is an object: a data frame is an object, a number is an object,... now a plot is an object too. Therefore, we can operate with it, in a especial way.

We begin calling the library (install it if you haven't yet).

```
library(ggplot2)
```

A plot object can be created with the `ggplot()` function.

```
P <- ggplot()

class(P)
```

```
## [1] "gg"     "ggplot"
```

If we try to show , we won't see anything interesting.

```
P
```

That's because we have to fill it with:

- Data
- The axis
- Aesthetics
- Labelling
- Distribution on the board
- ...

ggplot2 belongs to the tidyverse environment, therefore it is especially appropriate working with data frames. The data frames or tibbles will be the objects we'll use for providing the plot with data. Then we'll use the rest of the ggplot2 grammar to take advantage of this data and use it for visualizing.

4.1.2 Scatter plot

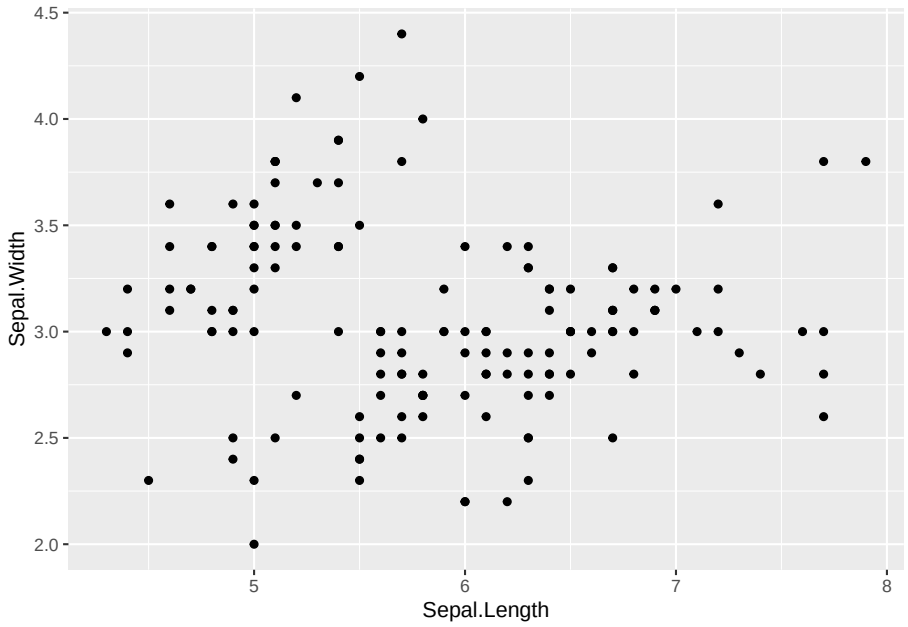
The scatter plot consists on several points on the plane. It is the most typical type plot for representing two dimensions from continuous (numerical) data: height vs. weight, income vs. age, ...

We've already seen that `ggplot()` generates a plot object. It can also receive the data frame we'll take the info from. But still, only indicating the data frame is not enough to plot anything.

```
ggplot(iris)
```

We must indicate specific info depending on the type of plot. For scatter plots, we add a layer of points with `geom_point()`, saying what the x axis represents and what the y axis represents.

```
ggplot(iris) +
  geom_point(aes(x = Sepal.Length, y = Sepal.Width))
```



We can also relate more things with the available data. For instance, we can color each point depending on some other column from the data frame.

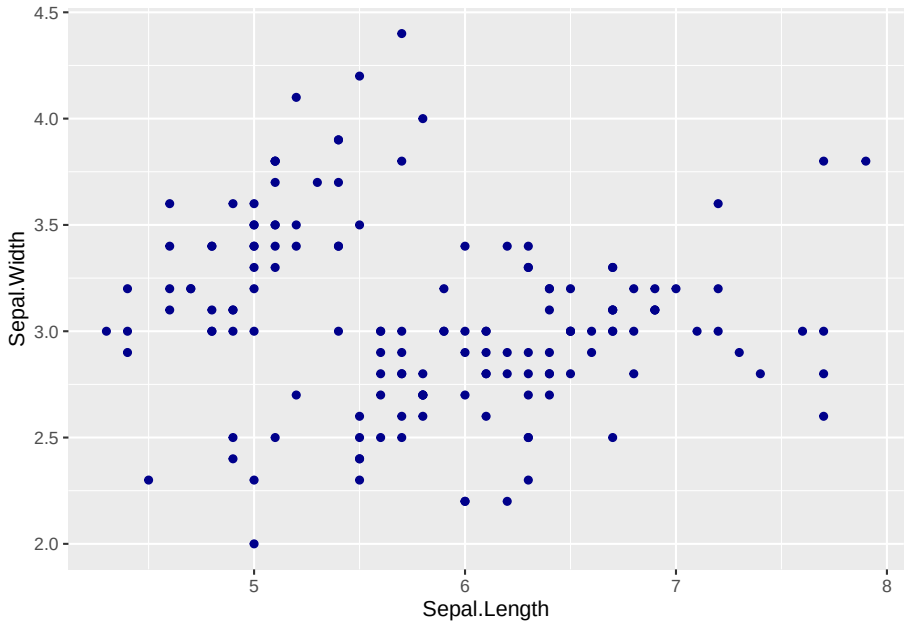
```
ggplot(iris) +
  geom_point(aes(x = Sepal.Length, y = Sepal.Width, colour = Species))
```

Notice that all this info we have used (x axis, y axis and colour) is inside the `aes()` function. If we tried to write this info outside the function we would get an error.

```
ggplot(iris) +
  geom_point(aes(x = Sepal.Length, y = Sepal.Width), colour = Species)
# Error in layer(data = data, mapping = mapping, stat = stat, geom = GeomPoint, : obje
```

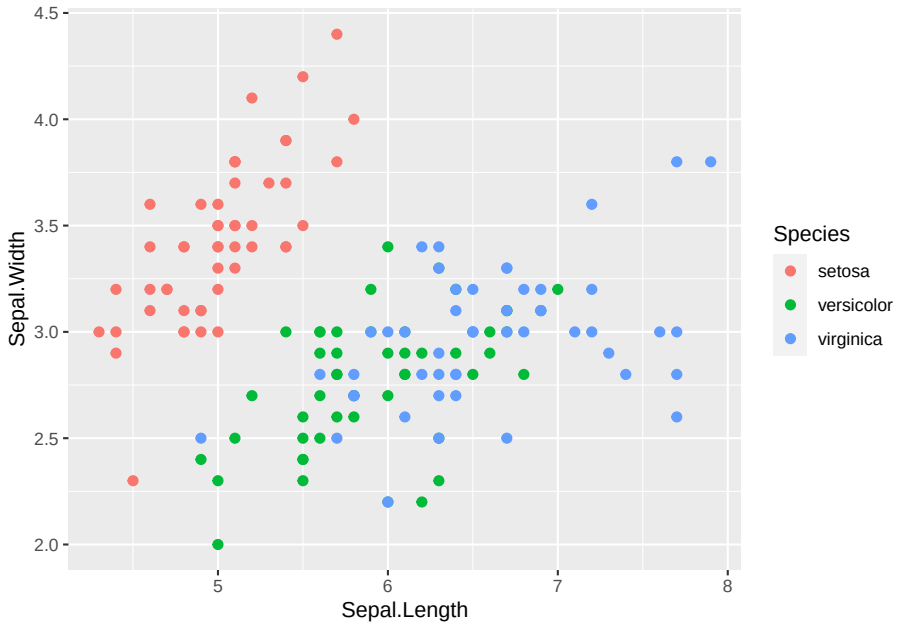
Outside `aes()` we can only include information for the graphic that doesn't depend on the data. For instance, if we wanted to colour all the points equally:

```
ggplot(iris) +
  geom_point(aes(x = Sepal.Length, y = Sepal.Width), colour = "darkblue")
```



Or increase its size:

```
ggplot(iris) +
  geom_point(aes(x = Sepal.Length, y = Sepal.Width, colour = Species), size = 2)
```



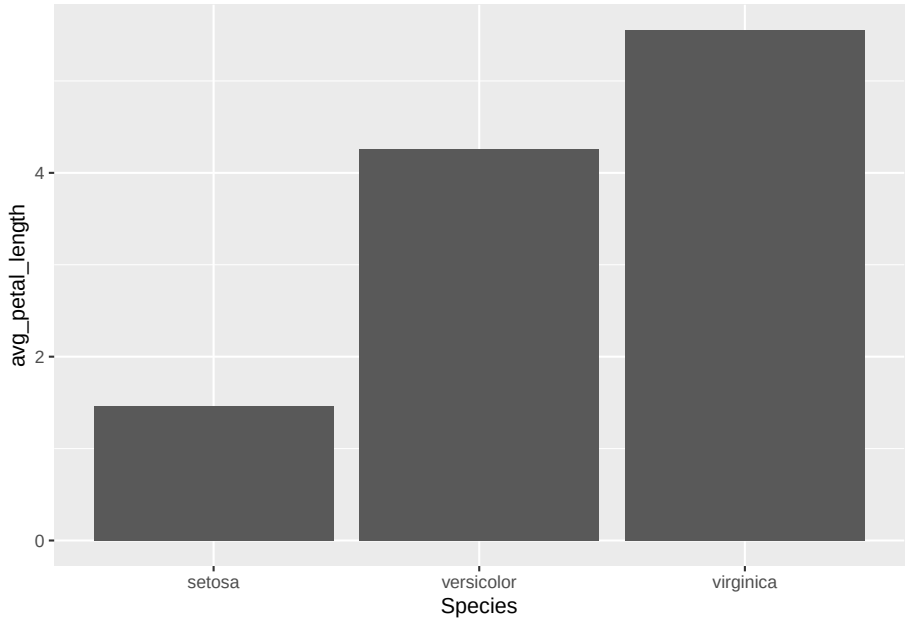
4.1.3 Columns

If we are working with just one numerical dimension, points may not be the best choice. Imagine you want to compare the average value of some metric at different groups. Plots or bars are good alternatives.

```
library(dplyr)

iris2plot <- iris %>%
  group_by(Species) %>%
  summarise(avg_petal_length = mean(Petal.Length),
            avg_petal_width = mean(Petal.Width))

iris2plot
```

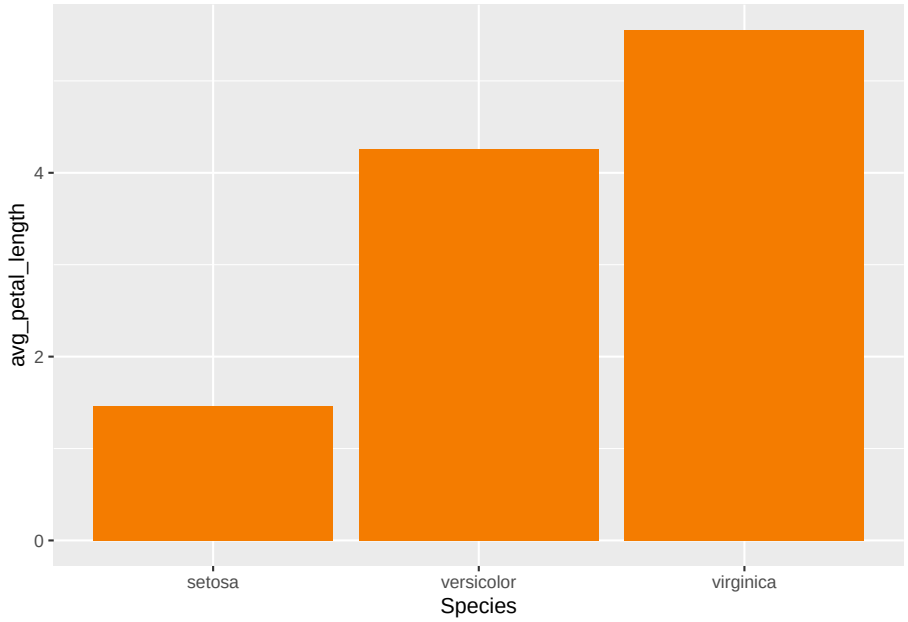
```
## # A tibble: 3 x 3
##   Species    avg_petal_length avg_petal_width
##   <fct>                 <dbl>           <dbl>
## 1 setosa                 1.46           0.246
## 2 versicolor             4.26           1.33
## 3 virginica              5.55           2.03
```

```
ggplot(iris2plot) +
  geom_col(aes(x = Species, y = avg_petal_length))
```



Similarly to the previous situation, you can decide whether extracting information from the data frame or using the same parameters for all the columns.

```
ggplot(iris2plot) +
  geom_col(aes(x = Species, y = avg_petal_length),
           fill = "#F47C00")
```



4.1.4 Lines

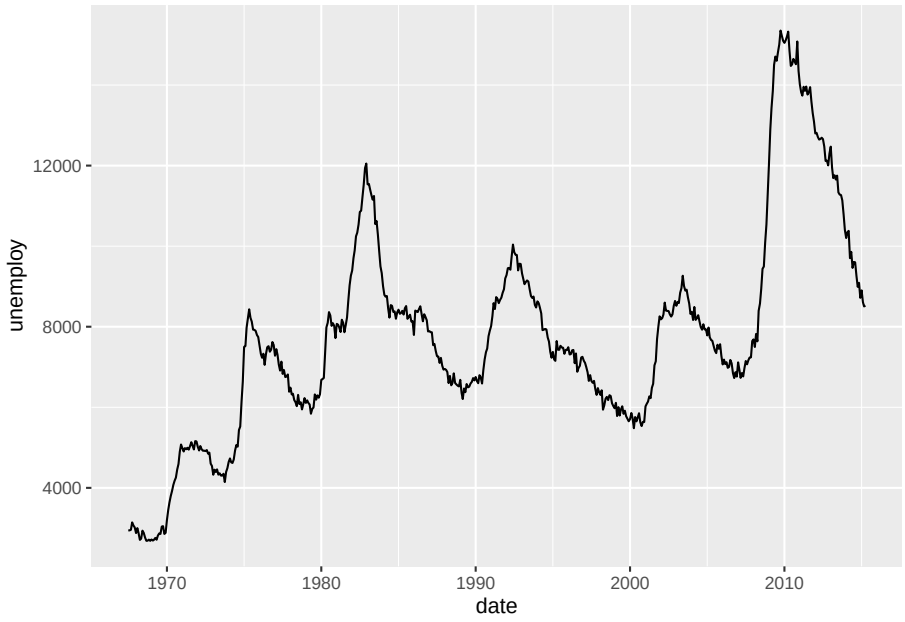
Line plots are associated with time series. Let's have a look at the economic dataset (details with `? economics`).

```
glimpse(economics)
```

```
## Rows: 574
## Columns: 6
## $ date     <date> 1967-07-01, 1967-08-01, 1967-09-01, 1967-10-01, 1967-11-01, ~
## $ pce      <dbl> 506.7, 509.8, 515.6, 512.2, 517.4, 525.1, 530.9, 533.6, 544.3~
## $ pop      <dbl> 198712, 198911, 199113, 199311, 199498, 199657, 199808, 19992~
## $ psavert  <dbl> 12.6, 12.6, 11.9, 12.9, 12.8, 11.8, 11.7, 12.3, 11.7, 12.3, 1~
## $ uempmed  <dbl> 4.5, 4.7, 4.6, 4.9, 4.7, 4.8, 5.1, 4.5, 4.1, 4.6, 4.4, 4.4, 4~
## $ unemploy <dbl> 2944, 2945, 2958, 3143, 3066, 3018, 2878, 3001, 2877, 2709, 2~
```

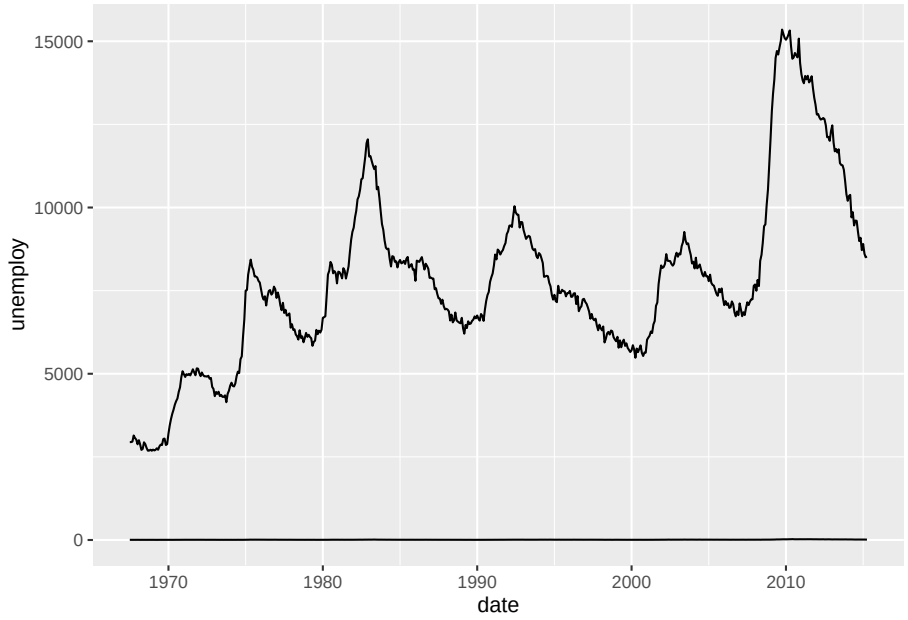
When using a x axis of `Date` class, ggplot2 knows how to work with it and will try to find proper labels for the axis.

```
# Notice that you can include the aes() functions inside geom_line() or ggplot()

ggplot(economics, aes(date, unemploy)) +
  geom_line()
```



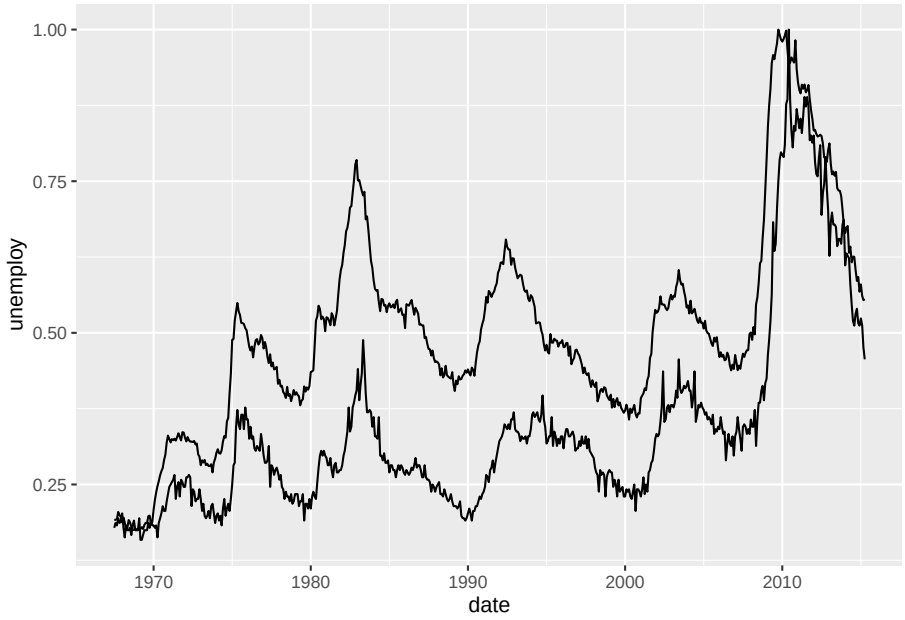
Now we want to plot the median duration of unemployment (`unempmed`) along with the unemployment. With ggplot2 we just have to add more layers.

```
ggplot(economics) +
  geom_line(aes(date, unemploy)) +
  geom_line(aes(date, uempmed))
```



The problem here is that the scales of the variables are too different. One solution for solving this is rescaling the data. Google Trends, for example, divides all the data by the maximum. For transforming the data we use dplyr.

```
scaled_economics <- economics %>%
  select(date, unemploy, uempmed) %>%
  mutate(unemploy = unemploy / max(unemploy),
         uempmed = uempmed / max(uempmed))

ggplot(scaled_economics) +
  geom_line(aes(date, unemploy)) +
  geom_line(aes(date, uempmed))
```



We should use different colors for each series. And include a legend. Too much work. Imagine if we had to plot all the series from the data frame. Too much code. Let's simplify this.

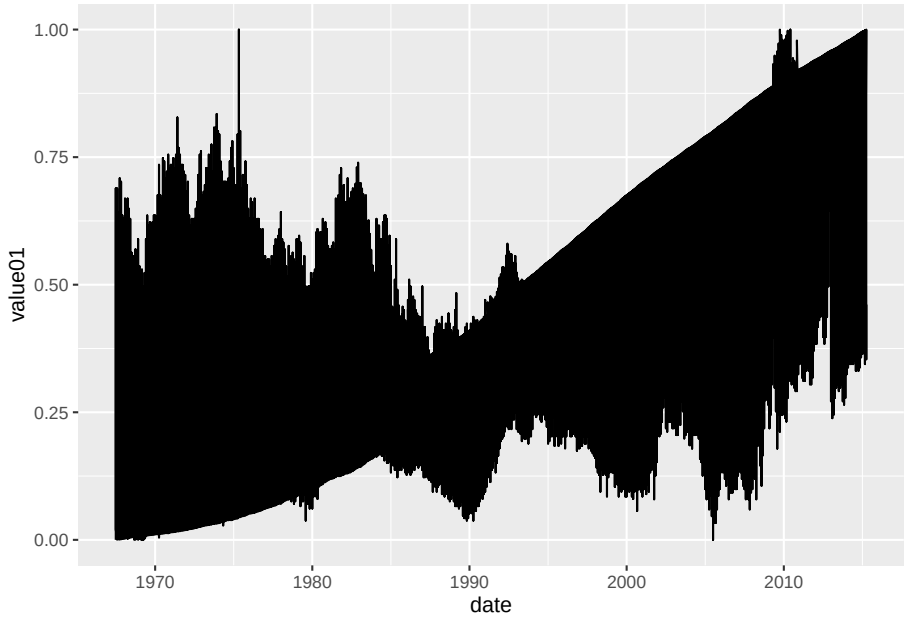
We need the data in a different format. Have a look at the `economics_long` data frame. It contains the same information as the previous one but it has more rows and less columns. We'll learn how to make these changes to a data frame later on but for now it is enough just using this one already prepared.

```
glimpse(economics_long)
```

```
## Rows: 2,870
## Columns: 4
## $ date     <date> 1967-07-01, 1967-08-01, 1967-09-01, 1967-10-01, 1967-11-01, ~
## $ variable <chr> "pce", "pce", "pce", "pce", "pce", "pce", "pce", "pce", "pce"~
## $ value    <dbl> 506.7, 509.8, 515.6, 512.2, 517.4, 525.1, 530.9, 533.6, 544.3~
## $ value01  <dbl> 0.0000000000, 0.0002652497, 0.0007615234, 0.0004706043, 0.000~
```

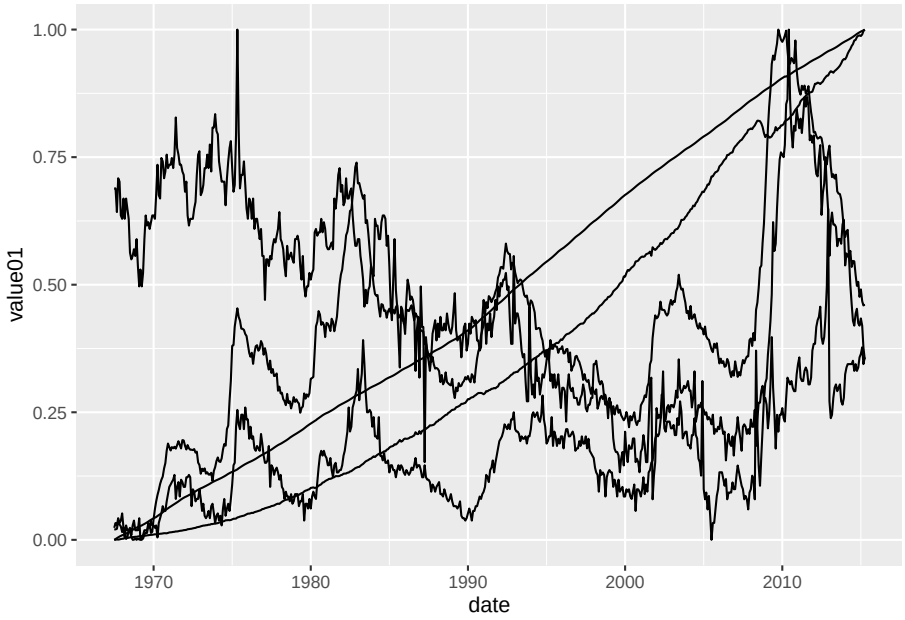
On the x axis we still want the date. On the y axis, the value that each variable gets. But if just write this, we will see that something is not working correctly.

```
ggplot(economics_long) +
  geom_line(aes(x = date, y = value01))
```



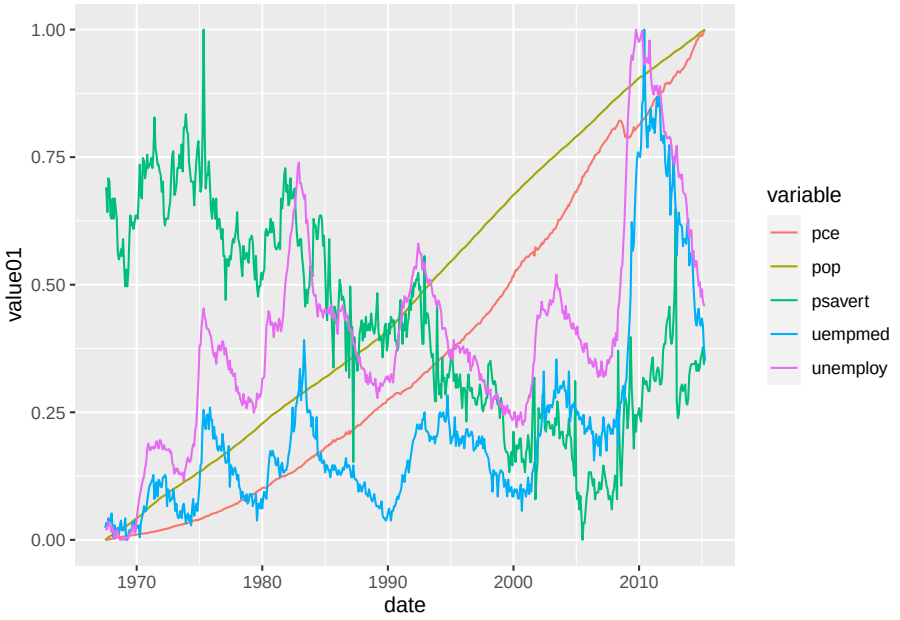
For obtaining the separation we want, we need to indicate that the values are separated in variables. We can do this with the `group=` parameter.

```
ggplot(economics_long) +
  geom_line(aes(x = date, y = value01, group = variable))
```



Or directly with the `colour=` parameter, which also provides us with a legend, mandatory in such a case.

```
ggplot(economics_long) +
  geom_line(aes(x = date, y = value01, colour = variable))
```



4.2 Exercises about scatter plots

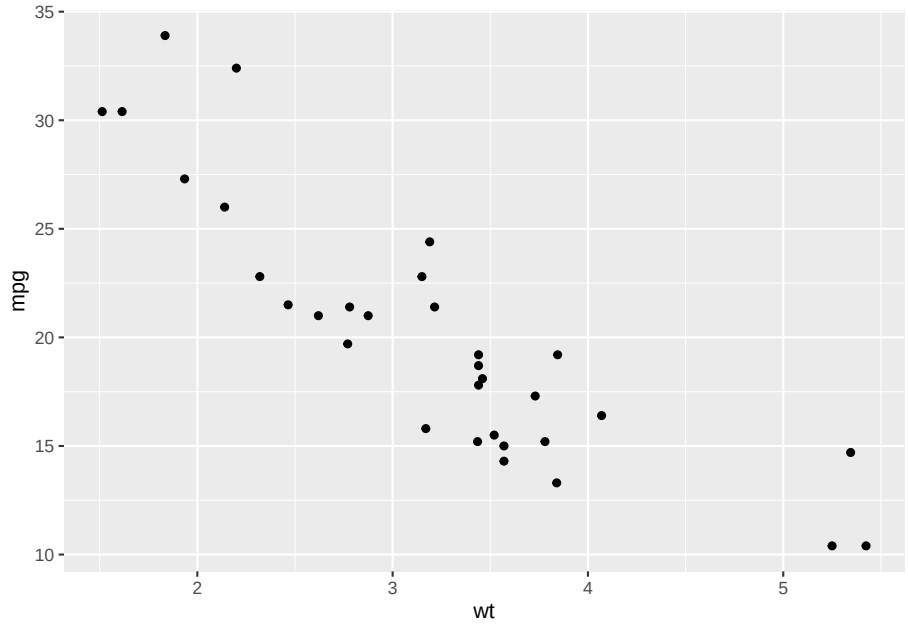
```
library(ggplot2)
library(dplyr)
```

4.2.1 Exercise 1

4.2.1.1 1a

Have a look at the `mtcars` data frame with `glimpse()` and `? mtcars`. Then create a scatter plot of the miles per gallon (on the y axis) and the weight (on the x axis). For creating the plot, just fill in the dots in the next code.

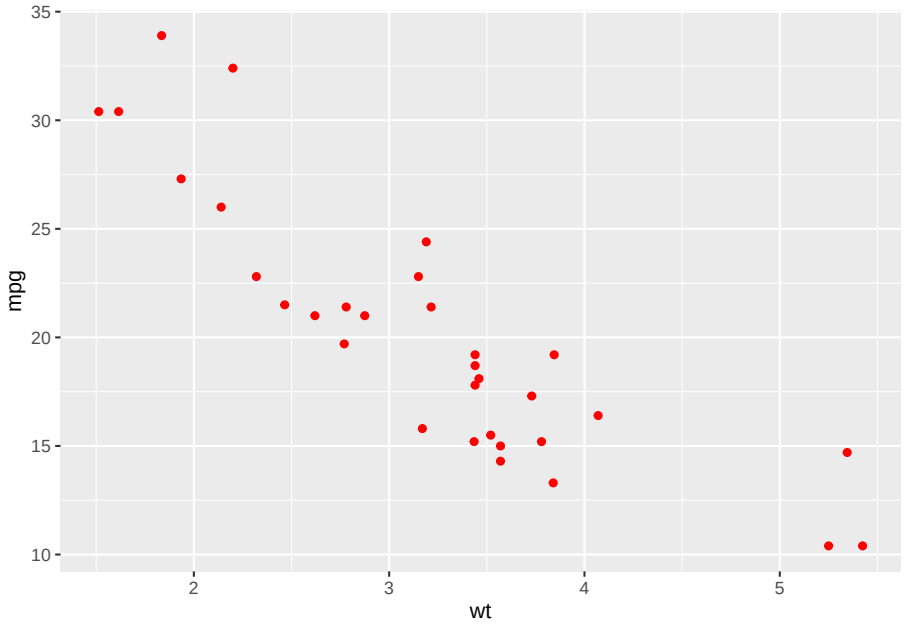
```
ggplot(...) +
  geom_point(aes(x = wt, y = mpg))
```



4.2.1.2 1b

Repeat the previous plot, colouring all the points from the previous plot in red. Notice that since the color doesn't depend on the data frame, it is typed outside the `aes()` function.

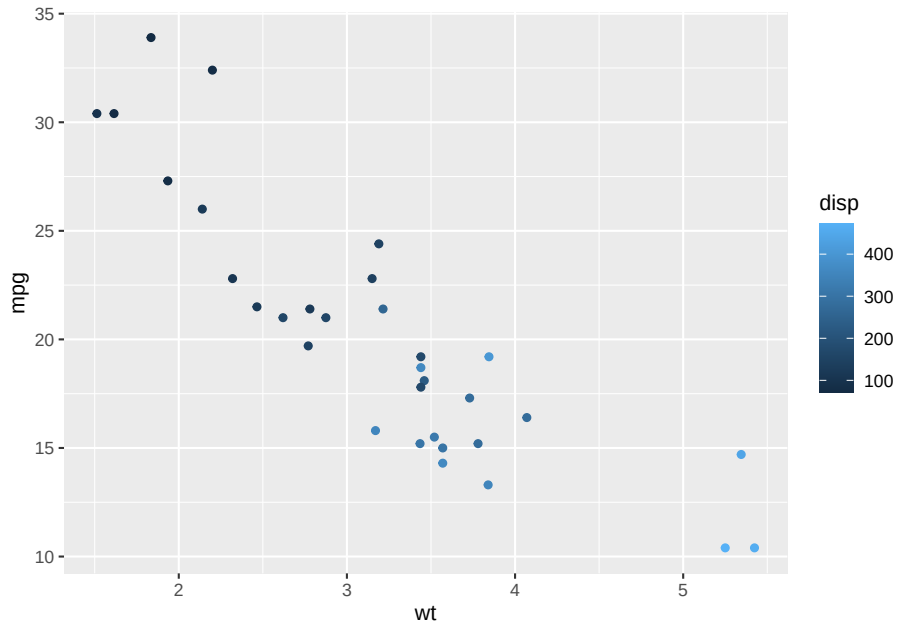
```
ggplot(...) +
  geom_point(aes(...), colour = "red")
```



4.2.1.3 1c

Repeat the same plot but colouring the dots based on their displacement value (`disp` column).

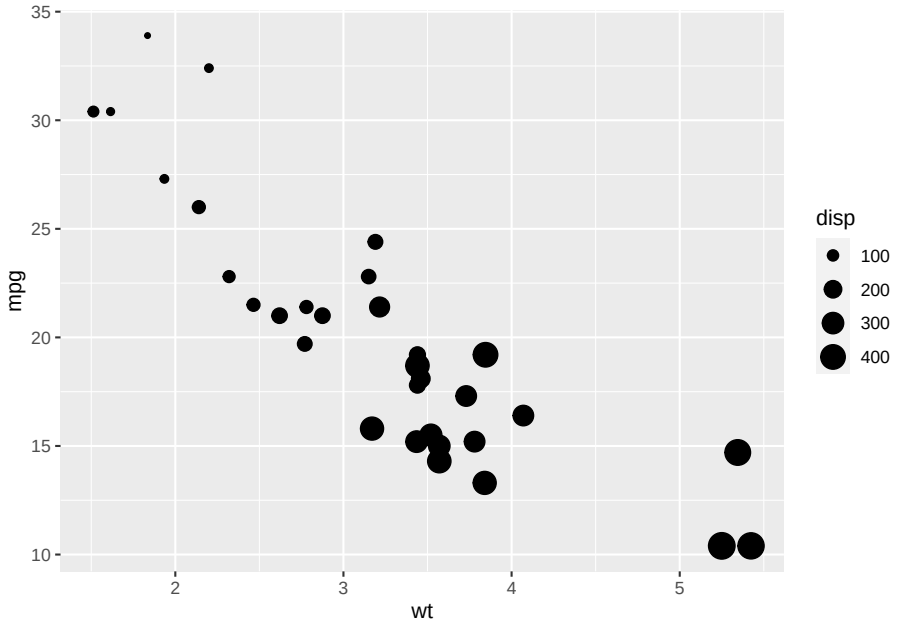
```
ggplot(...) +
  geom_point(aes(x = ..., y = ..., colour = ...))
```



4.2.1.4 1d

Instead of colouring the dots in different colours, make them have a different size depending again on the `disp` variable.

```
ggplot(...) +
  geom_point(aes(...))
```



4.2.2 Exercise 2

4.2.2.1 2a

Scatter plots can also be used for non continuous data. In R, when dealing with continuous data, we usually use the `numeric` class (weight, heights, incomes,...). But there are some variables that are written with numbers but aren't continuous (age, number of cylinders, identificators). What is the class of the number of cylinders in the `mtcars` data frame? Remember to use `? mtcars` if you don't know the name of the column in the data frame.

4.2.2.2 2b

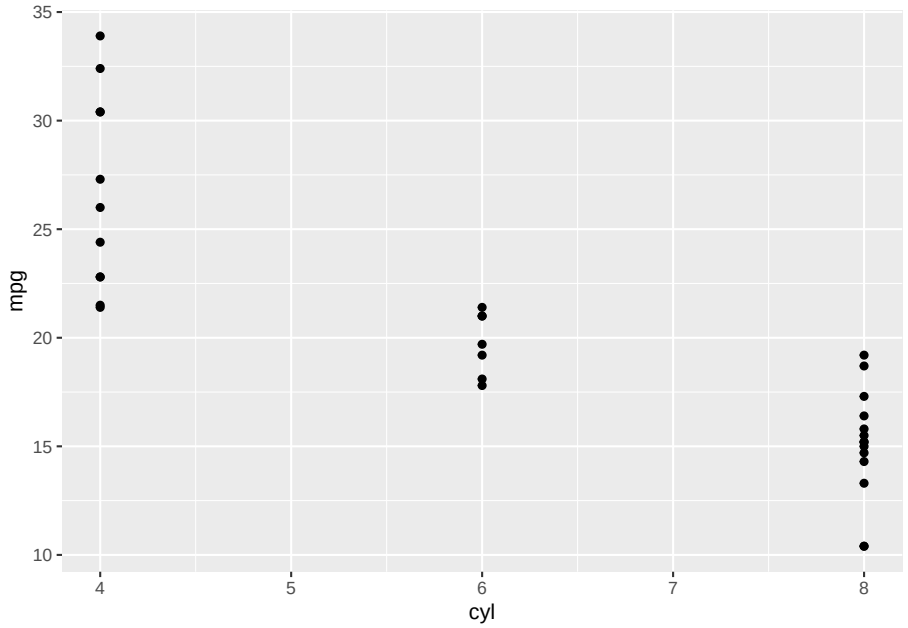
What are its unique values. Fill in the gap:

```
mtcars %>% distinct(...)
```

4.2.2.3 2c

Based on the distinct values, the class of the column it's not the best. Try to plot a scatter plot the miles per gallon (y axis) versus the number of cylinders.

```
ggplot(mtcars) +
  geom_point(aes(...))
```



In general, nothing wrong. But the x axis shows values that are not possible (5 and 7). This is because the number of cylinders should be a categorical variable and not a numerical one. Let's fix it.

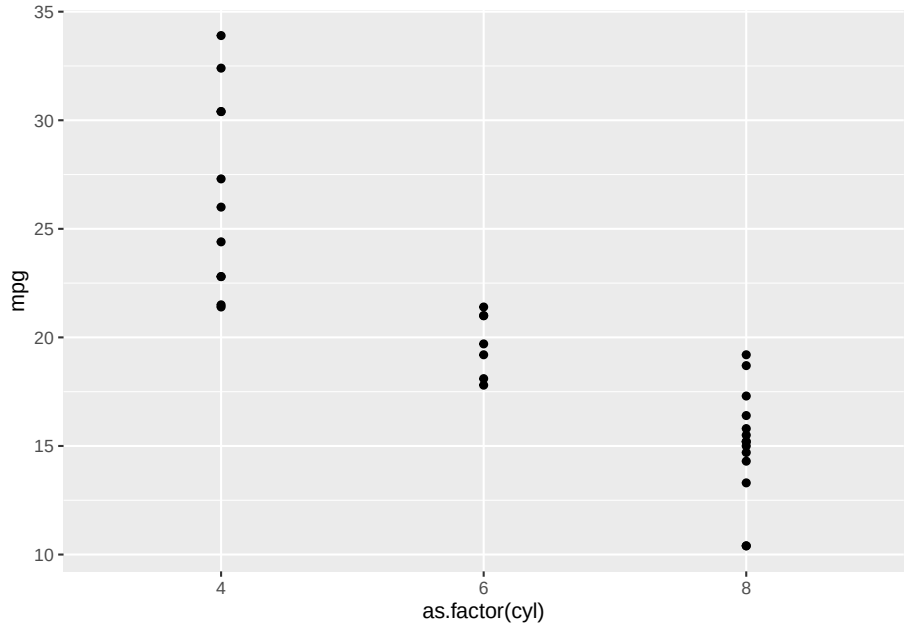
4.2.2.4 2d

In R, categorial variables belong to a new class: `factor`. We can easily convert some variables to the factor class with the `as.factor()` function.

```
mtcars <- mtcars %>%
  mutate(cyl = as.factor(cyl))
```

Fill in the gaps for plotting the same scatter plot as before and compare the result.

```
ggplot(...) +
  ...
```



4.2.3 Exercise 3

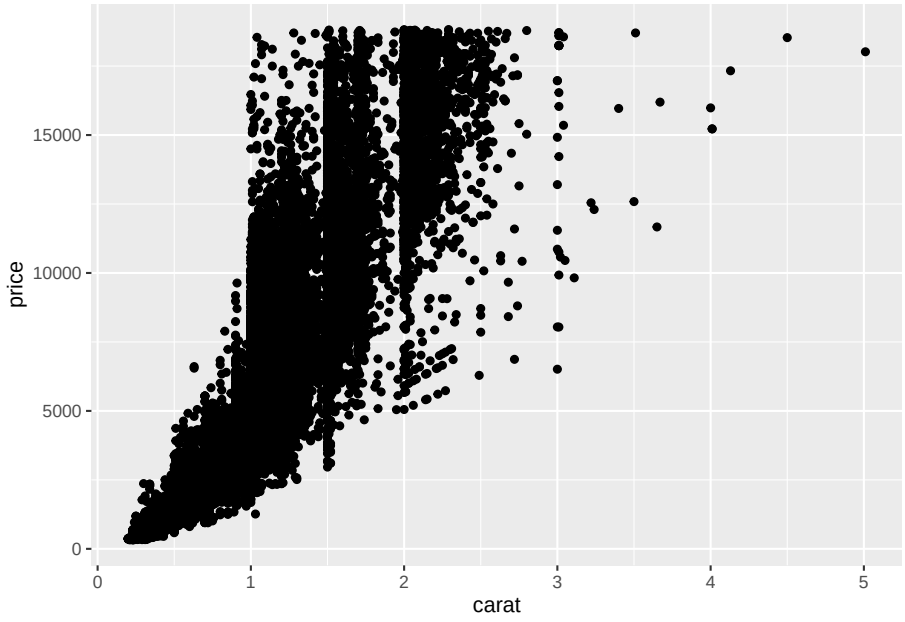
4.2.3.1 3a

With ggplot2 you have available the `diamonds` data frame. Have a look at its info with `? diamonds` and `glimpse(diamonds)`.

4.2.3.2 3b

Create a scatter plot with the price of the diamonds (y axis) vs the weight of the diamond (x axis). There are more than 50,000 diamonds, so it may take longer than expected plotting the plot.

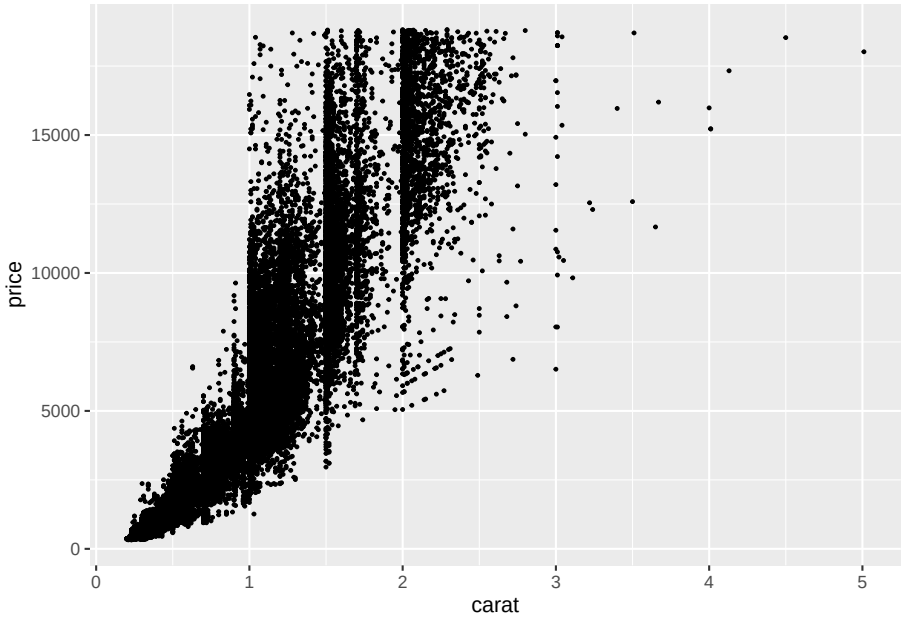
```
ggplot(diamonds) +
  ...(aes(...))
```



4.2.3.3 3c

There are too many dots, therefore they overlap and the result is a cloud of data not very comfortable to look at. Let's fix it. First, reduce the size of all the points. Try with `size = 0.5`.

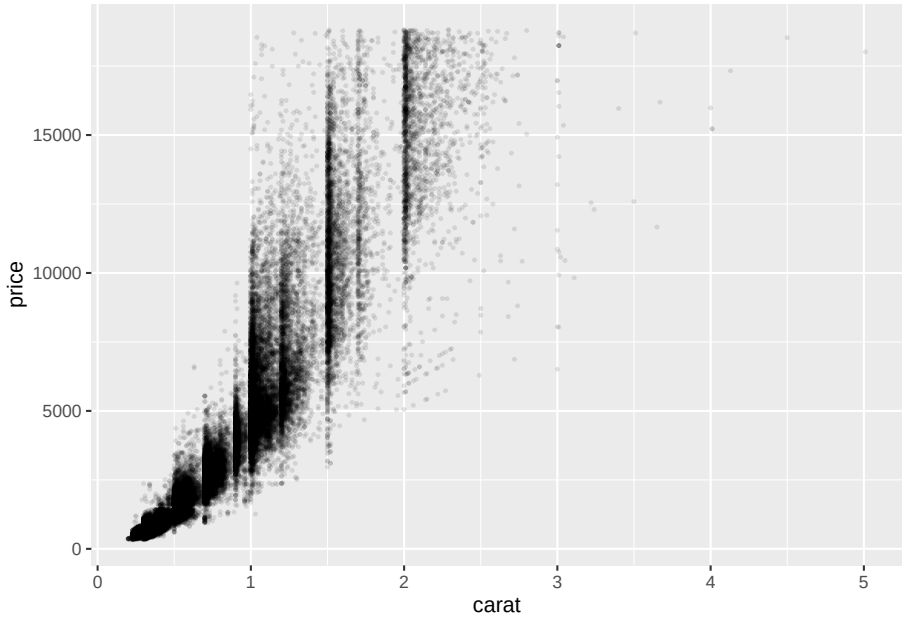
```
ggplot(...) +
  ...(aes(...), ...)
```



4.2.3.4 3d

One more thing to improve it. We can give some transparency to the points. This will help us seeing where there are too many points together. The parameter used for this is called `alpha` (very common not only in R but also in other programming languages that deal with color, such as CSS). This parameter takes values from 0 (invisible) to 1 (opaque). Set this value to 0.1, alongside the already defined size. This time, you write the code from scratch, but remember that copy-pasting is key when programming.

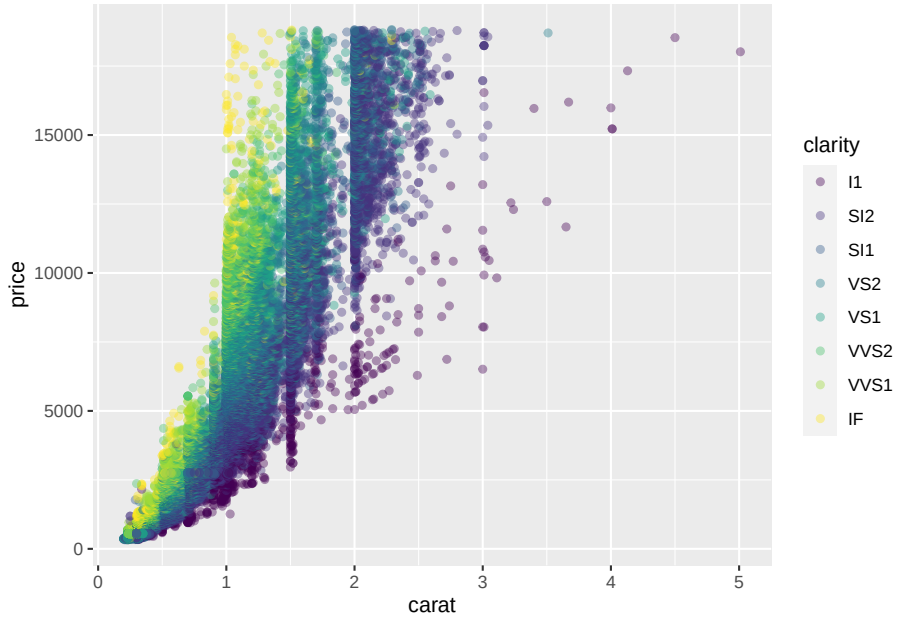
. . .



4.2.3.5 3e

Now we are giving some color to the plot. This color will be based on the clarity of the diamonds, which is a column from the data frame. We are setting the transparency to 0.4, to make that overlapping easier to see.

```
...
```



4.2.4 Exercise 4

Let us create a data frame from the `countries_of_the_world.csv` file you used for Assessment 1. We will use the `clean_names()` function from the janitor library to simplify the names of the columns. We are also removing all the rows with some `NA` value.

```
library(readr)
library(janitor)

df_countries <- read_csv("data/countries_of_the_world.csv", locale = locale(decimal_mark = ","))
  clean_names() %>%
  na.omit()
```

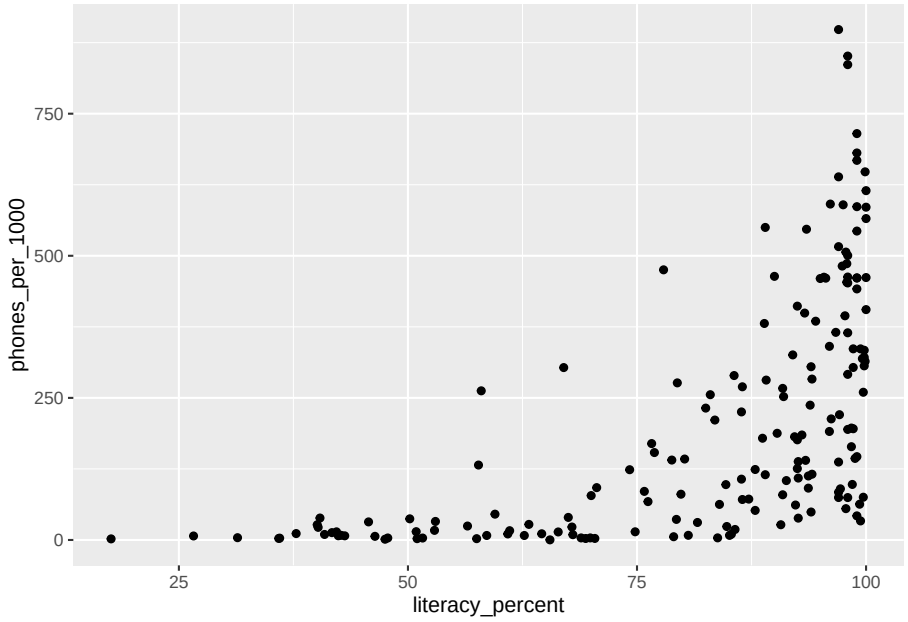
```
##
## -- Column specification --------------------------------------------------------
## cols(
##   .default = col_double(),
##   Country = col_character(),
##   Region = col_character()
## )
## i Use `spec()` for the full column specifications.
```

```
glimpse(df_countries)
```

```
## Rows: 179
## Columns: 20
## $ country                           <chr> "Afghanistan", "Albania", "Algeria", ~
## $ region                            <chr> "ASIA (EX. NEAR EAST)", "EASTERN EURO~
## $ population                        <dbl> 31056997, 3581655, 32930091, 13477, 6~
## $ area_sq_mi                        <dbl> 647500, 28748, 2381740, 102, 443, 276~
## $ pop_density_per_sq_mi             <dbl> 48.0, 124.6, 13.8, 132.1, 156.0, 14.4~
## $ coastline_coast_area_ratio        <dbl> 0.00, 1.26, 0.04, 59.80, 34.54, 0.18,~
## $ net_migration                     <dbl> 23.06, -4.93, -0.39, 10.76, -6.15, 0.~
## $ infant_mortality_per_1000_births <dbl> 163.07, 21.52, 31.00, 21.03, 19.46, 1~
## $ gdp_per_capita                    <dbl> 700, 4500, 6000, 8600, 11000, 11200, ~
## $ literacy_percent                  <dbl> 36.0, 86.5, 70.0, 95.0, 89.0, 97.1, 9~
## $ phones_per_1000                   <dbl> 3.2, 71.2, 78.1, 460.0, 549.9, 220.4,~
## $ arable_percent                    <dbl> 12.13, 21.09, 3.22, 0.00, 18.18, 12.3~
## $ crops_percent                     <dbl> 0.22, 4.42, 0.25, 0.00, 4.55, 0.48, 2~
## $ other_percent                     <dbl> 87.65, 74.49, 96.53, 100.00, 77.27, 8~
## $ climate                           <dbl> 1.0, 3.0, 1.0, 2.0, 2.0, 3.0, 4.0, 2.~
## $ birthrate                         <dbl> 46.60, 15.11, 17.14, 14.17, 16.93, 16~
## $ deathrate                         <dbl> 20.34, 5.22, 4.61, 5.34, 5.37, 7.55, ~
## $ agriculture                       <dbl> 0.380, 0.232, 0.101, 0.040, 0.038, 0.~
## $ industry                          <dbl> 0.240, 0.188, 0.600, 0.180, 0.220, 0.~
## $ service                           <dbl> 0.380, 0.579, 0.298, 0.780, 0.743, 0.~
```

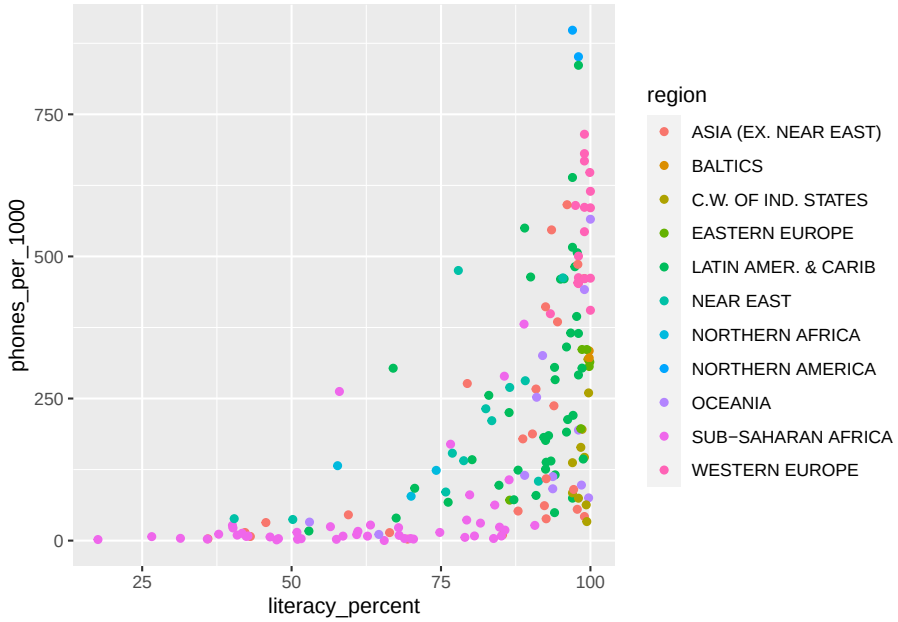
4.2.4.1 4a

Plot the `phones_per_1000` columns against the `literacy_percent` column, with a scatter plot.



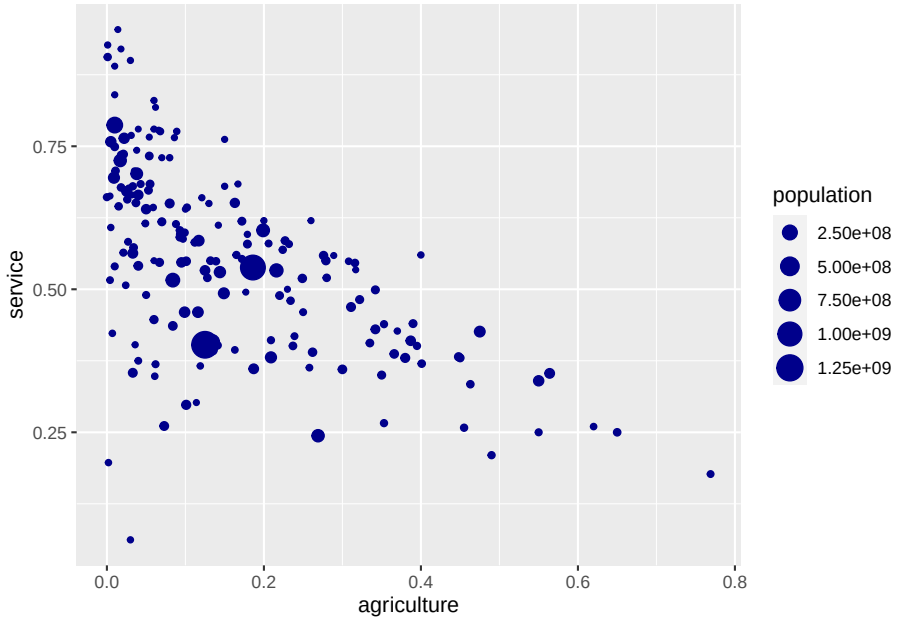
4.2.4.2 4b

In the same plot, colour each point based on its region.



4.2.4.3 4c

Make scatter plot of the `service` columns vs. the `agriculture` column. Change the size of each point depending on each population and color all of them in `"darkblue"`.



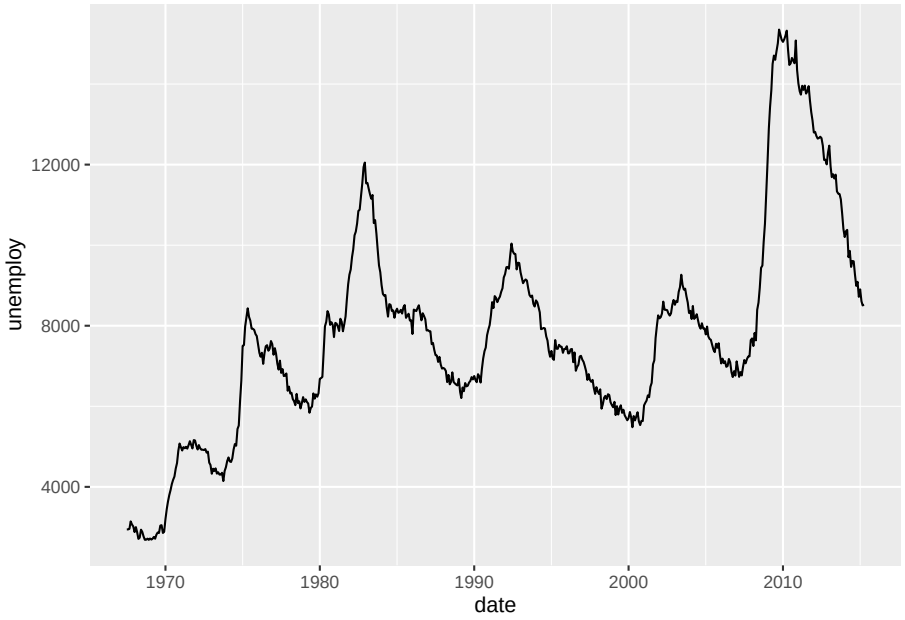
4.3 Exercises about line plots

```
library(ggplot2)
library(dplyr)
```

4.3.1 Exercise 1

4.3.1.1 1a

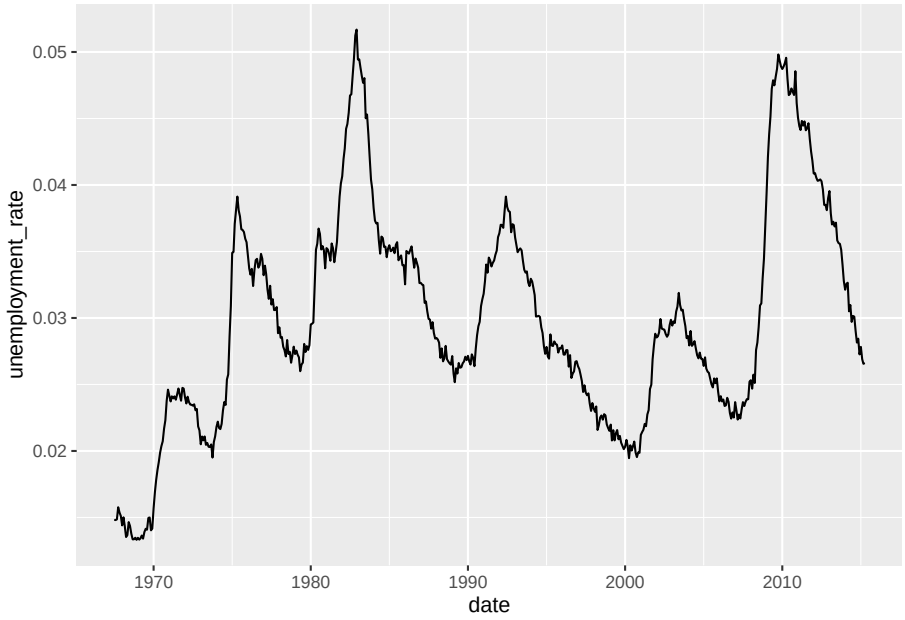
From the `economics` data frame plot `unemploy` vs `date` using a line plot. You will need `geom_line()`.



4.3.1.2 1b

Using dplyr, calculate a new column as the unemployment rate (umemployment divided by the population). Plot this column as a time series, in a similar way you did before.

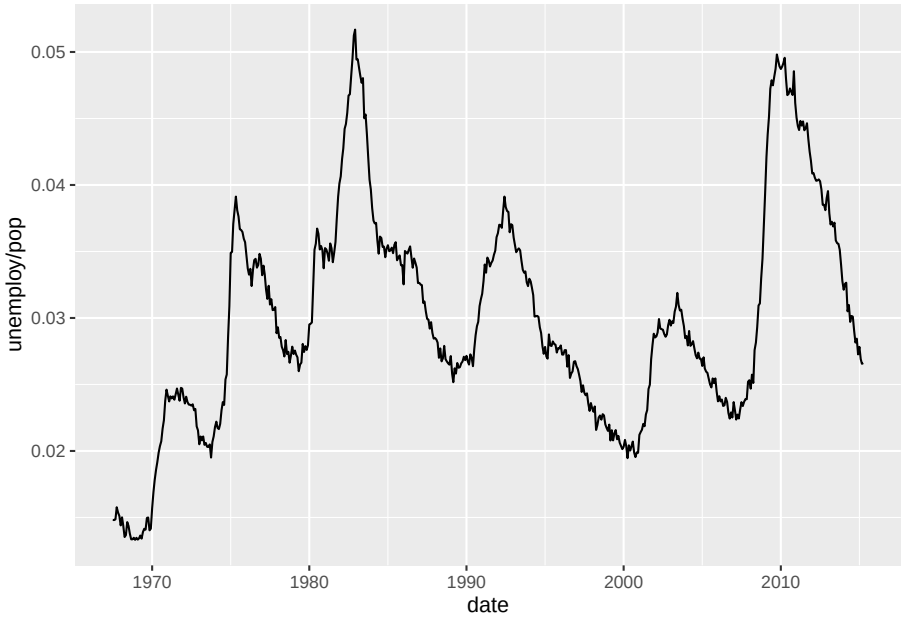
```
economics_new <- ... %>%
  ...(unemployment_rate = ...)

# Code for the plot:
...
```



4.3.1.3 1c

Something comfortable about ggplot2 is that it is also able to make some calculations. For intance, in the previous plot, it wasn't necessary that dplyr step. We can calculate the unemployment rate on the fly. Try to use the formula `umemploy / pop` when specifying the y axis.

```
... +
  ...(...(x = ..., y = ...))
```

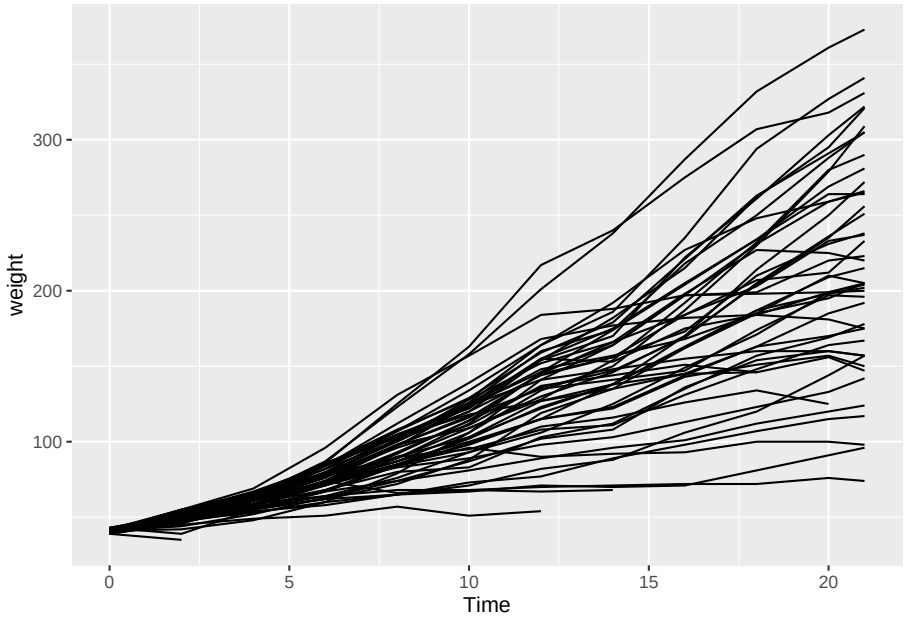


4.3.2 Exercise 2

4.3.2.1 2a

From the `ChickWeight` data frame, create a lines plot with the evolution over Time of the `weight` variable. Take into account that we need a different line for each `Chick`. For achieving this, map the `Chick` column to the `group=` parameter, inside the `aes()` function.

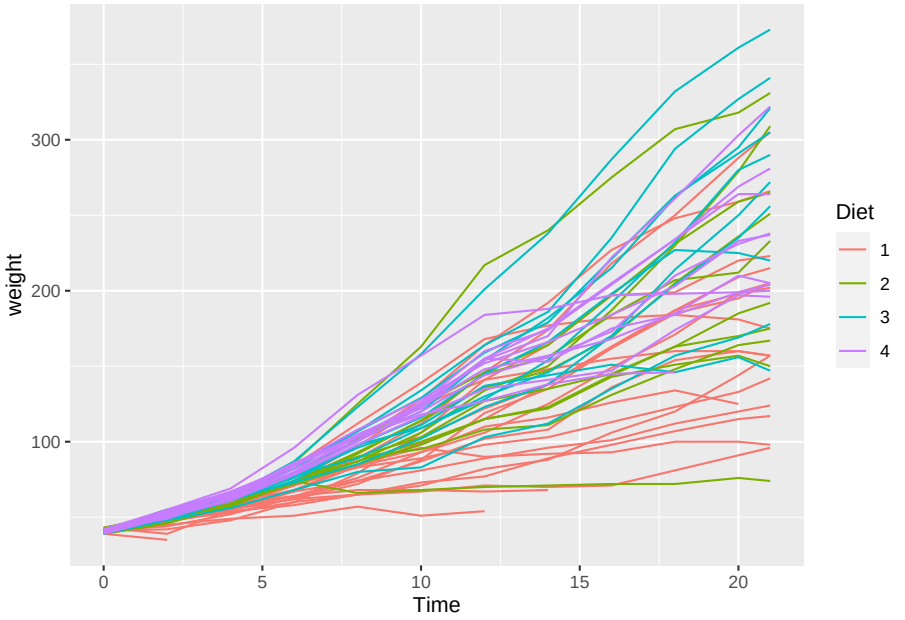
```
... +
  ...(aes(x = ..., y = ..., group = ...))
```



4.3.2.2 2b

Now colour each line depending on the `Diet` the chick has received.

```
...
```



4.4 Columns and boxplots

4.5 Preparing the data

We will study a breast cancer dataset, very common on data analysis. It contains several cases of benign and malign cases of breast cancer with information about the biopsies. It is often used as an example for supervised modelling: the goal would be predicting whether the case is benign or malign from the rest of the variables.

We won't get through the modelling steps but we will see how to get some insights about the relation among the explanatory variables and the *target* variable.

```
library(dplyr)
library(ggplot2)
library(readr)
library(janitor)
```

We read the csv file and make some name cleaning.

```
df_cancer <- read_csv("data/breast_cancer_data.csv", col_names = FALSE)
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   X1 = col_double(),
##   X2 = col_double(),
##   X3 = col_double(),
##   X4 = col_double(),
##   X5 = col_double(),
##   X6 = col_double(),
##   X7 = col_double(),
##   X8 = col_double(),
##   X9 = col_double(),
##   X10 = col_double(),
##   X11 = col_double()
## )
```

```
df_cancer <- clean_names(df_cancer)

glimpse(df_cancer)
```

```
## Rows: 699
## Columns: 11
## $ x1  <dbl> 1000025, 1002945, 1015425, 1016277, 1017023, 1017122, 1018099, 101~
## $ x2  <dbl> 5, 5, 3, 6, 4, 8, 1, 2, 2, 4, 1, 2, 5, 1, 8, 7, 4, 4, 10, 6, 7, 10~
## $ x3  <dbl> 1, 4, 1, 8, 1, 10, 1, 1, 1, 2, 1, 1, 3, 1, 7, 4, 1, 1, 7, 1, 3, 5,~
## $ x4  <dbl> 1, 4, 1, 8, 1, 10, 1, 2, 1, 1, 1, 1, 3, 1, 5, 6, 1, 1, 7, 1, 2, 5,~
## $ x5  <dbl> 1, 5, 1, 1, 3, 8, 1, 1, 1, 1, 1, 1, 3, 1, 10, 4, 1, 1, 6, 1, 10, 3~
## $ x6  <dbl> 2, 7, 2, 3, 2, 7, 2, 2, 2, 2, 1, 2, 2, 2, 7, 6, 2, 2, 4, 2, 5, 6, ~
## $ x7  <dbl> 1, 10, 2, 4, 1, 10, 10, 1, 1, 1, 1, 1, 3, 3, 9, 1, 1, 1, 10, 1, 10~
## $ x8  <dbl> 3, 3, 3, 3, 3, 9, 3, 3, 1, 2, 3, 2, 4, 3, 5, 4, 2, 3, 4, 3, 5, 7, ~
## $ x9  <dbl> 1, 2, 1, 7, 1, 7, 1, 1, 1, 1, 1, 1, 4, 1, 5, 3, 1, 1, 1, 1, 4, 10,~
## $ x10 <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 1, 4, 1, 1, 1, 2, 1, 4, 1, ~
## $ x11 <dbl> 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 4, 2, 4, 4, 2, 2, 4, 2, 4, 4, ~
```

```
names(df_cancer) <- c("id", "clump", "unif_size", "unif_shape", "adhesion",
                      "epithelial_size", "nuclei", "bland_chromatin",
                      "normal_nucleoli", "mitoses", "class")

glimpse(df_cancer)
```

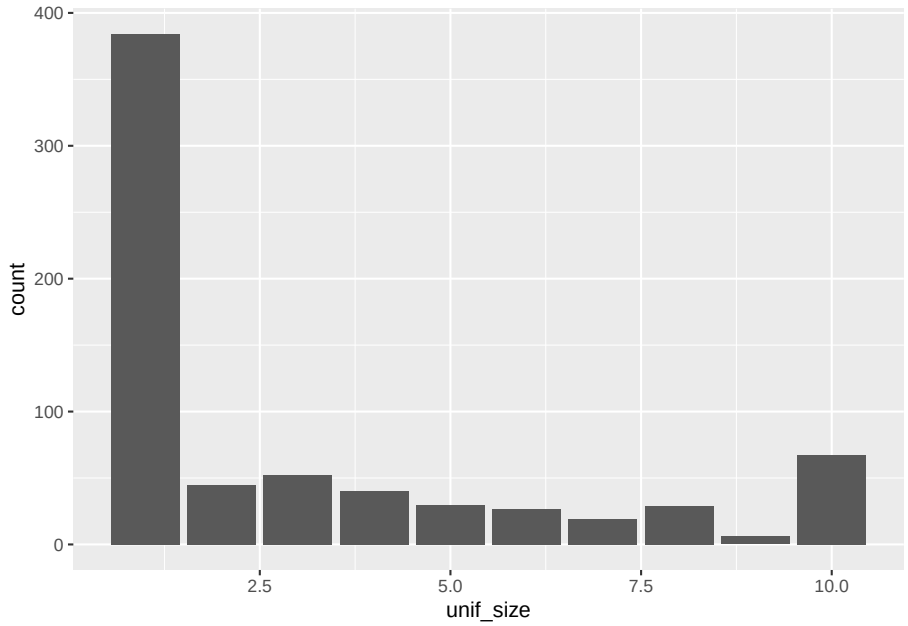
```
## Rows: 699
```

```
## Columns: 11
## $ id              <dbl> 1000025, 1002945, 1015425, 1016277, 1017023, 1017122, ~
## $ clump           <dbl> 5, 5, 3, 6, 4, 8, 1, 2, 2, 4, 1, 2, 5, 1, 8, 7, 4, 4, ~
## $ unif_size       <dbl> 1, 4, 1, 8, 1, 10, 1, 1, 1, 2, 1, 1, 3, 1, 7, 4, 1, 1,~
## $ unif_shape      <dbl> 1, 4, 1, 8, 1, 10, 1, 2, 1, 1, 1, 1, 3, 1, 5, 6, 1, 1,~
## $ adhesion        <dbl> 1, 5, 1, 1, 3, 8, 1, 1, 1, 1, 1, 1, 3, 1, 10, 4, 1, 1,~
## $ epithelial_size <dbl> 2, 7, 2, 3, 2, 7, 2, 2, 2, 2, 1, 2, 2, 2, 7, 6, 2, 2, ~
## $ nuclei          <dbl> 1, 10, 2, 4, 1, 10, 10, 1, 1, 1, 1, 1, 3, 3, 9, 1, 1, ~
## $ bland_chromatin <dbl> 3, 3, 3, 3, 3, 9, 3, 3, 1, 2, 3, 2, 4, 3, 5, 4, 2, 3, ~
## $ normal_nucleoli <dbl> 1, 2, 1, 7, 1, 7, 1, 1, 1, 1, 1, 1, 4, 1, 5, 3, 1, 1, ~
## $ mitoses         <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 1, 4, 1, 1, 1, ~
## $ class           <dbl> 2, 2, 2, 2, 2, 4, 2, 2, 2, 2, 2, 2, 4, 2, 4, 4, 2, 2, ~
```

4.6 Bar plots

Bar plots are comfortable for seeing whether a variable changes depending on the level. We can have a quick look at how cases there are for a variable, for instances, `unif_size`.

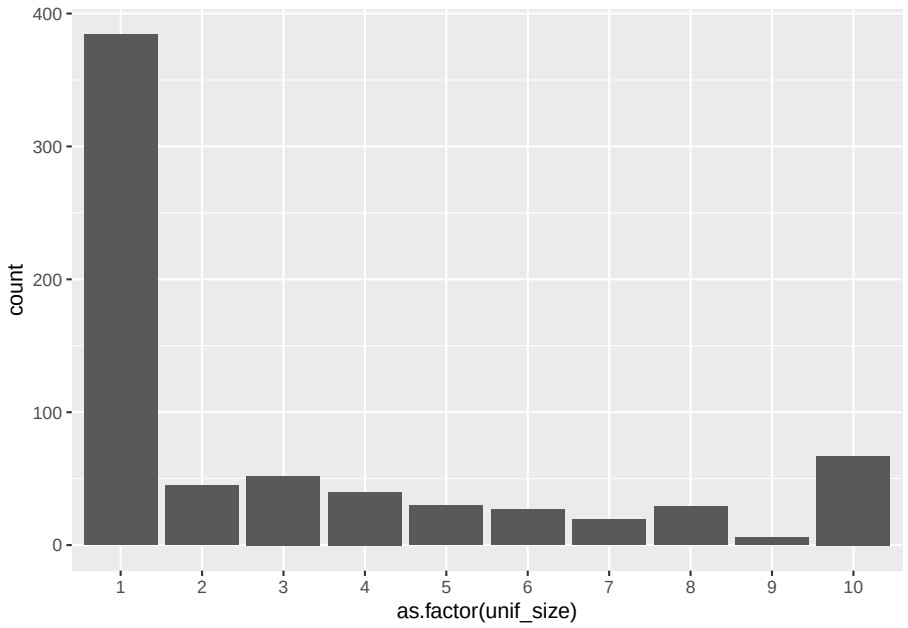
```
ggplot(df_cancer) +
  geom_bar(aes(x = unif_size))
```



Note that the x axis shows values as if the variable were numeric. In fact, it is categorical (this is easy to see if look up some documentation about the dataset

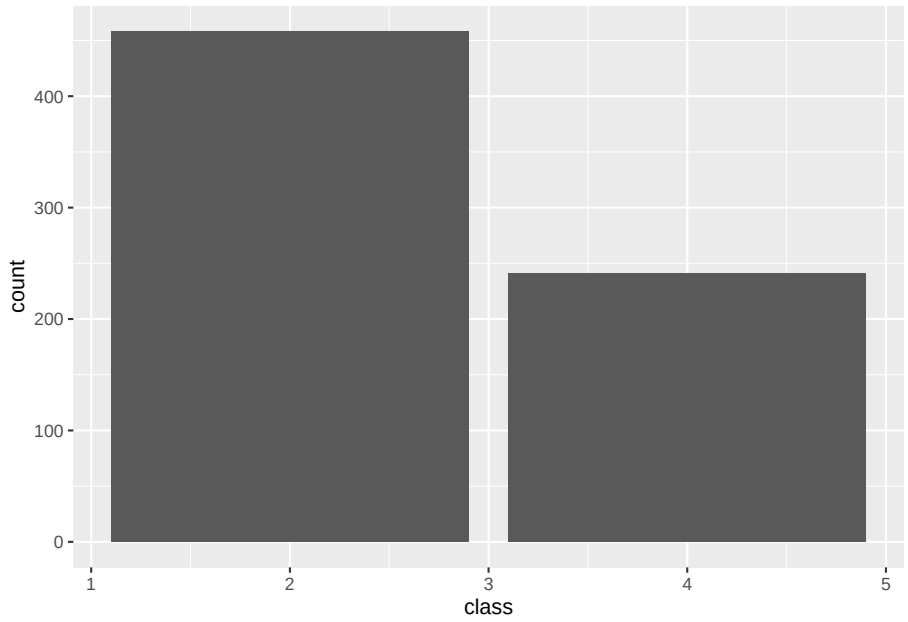
online), since there are ten classes, not ten numbers, even though these classes are represented by 1, 2, 3. . . . Since we don't have a dictionary for the classes, we can transform the variable directly into a categorical one, taking the numbers as the names. This can be done with dplyr but also with ggplot.

```
ggplot(df_cancer) +
  geom_bar(aes(x = as.factor(unif_size)))
```



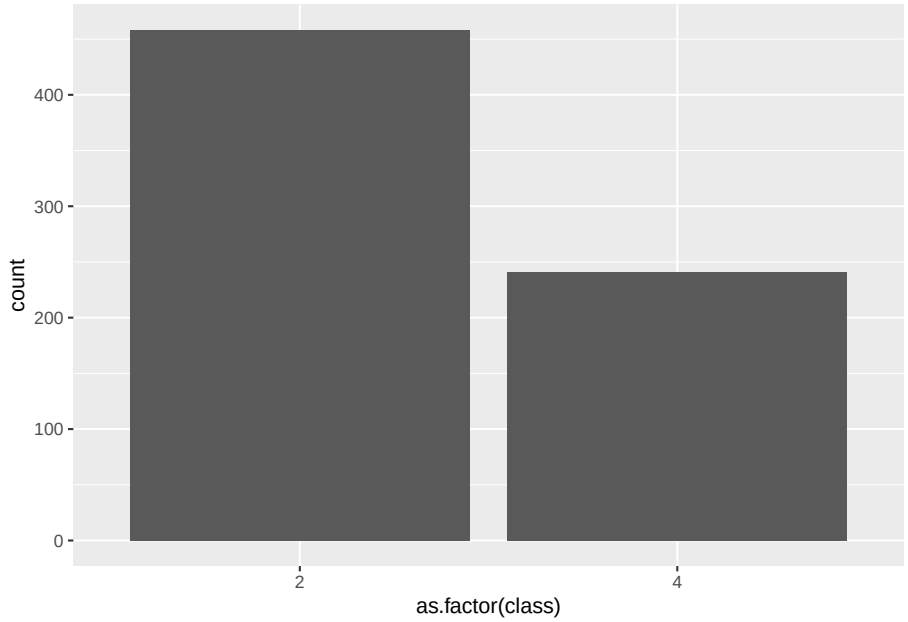
The same problem will we have if we try to plot the classes of the target variable.

```
ggplot(df_cancer) +
  geom_bar(aes(x = class))
```



Again, we change it to a `factor` variable.

```
ggplot(df_cancer) +
  geom_bar(aes(x = as.factor(class)))
```



Instead of working with the numeric labels, we can work with a `character` column and then create the `factor` column.

```
df_cancer_new <- df_cancer %>%
  mutate(class = if_else(class == 2, "benign", "malign"))

df_cancer_new %>%
  distinct(class)
```

```
## # A tibble: 2 x 1
##   class
##   <chr>
## 1 benign
## 2 malign
```

```
df_cancer_new <- df_cancer %>%
  mutate(class = factor(class))

df_cancer_new %>%
  distinct(class)
```

```
## # A tibble: 2 x 1
##   class
##   <fct>
## 1 2
## 2 4
```

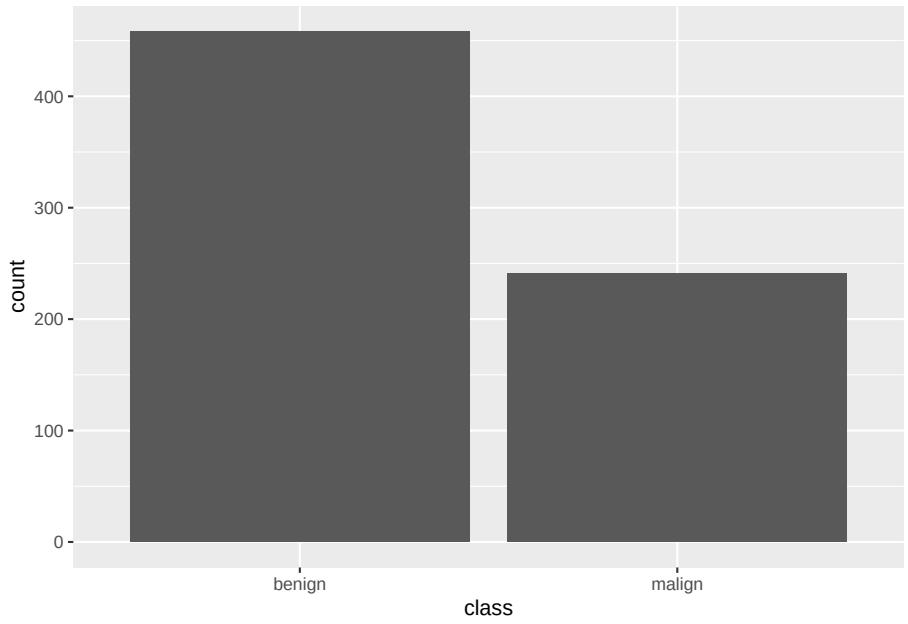
Other way around is defining directly the factor with the proper labels.

```
df_cancer_new <- df_cancer %>%
  mutate(class = factor(class, labels = c("benign", "malign")))

df_cancer_new %>%
  distinct(class)
```

```
## # A tibble: 2 x 1
##   class
##   <fct>
## 1 benign
## 2 malign
```

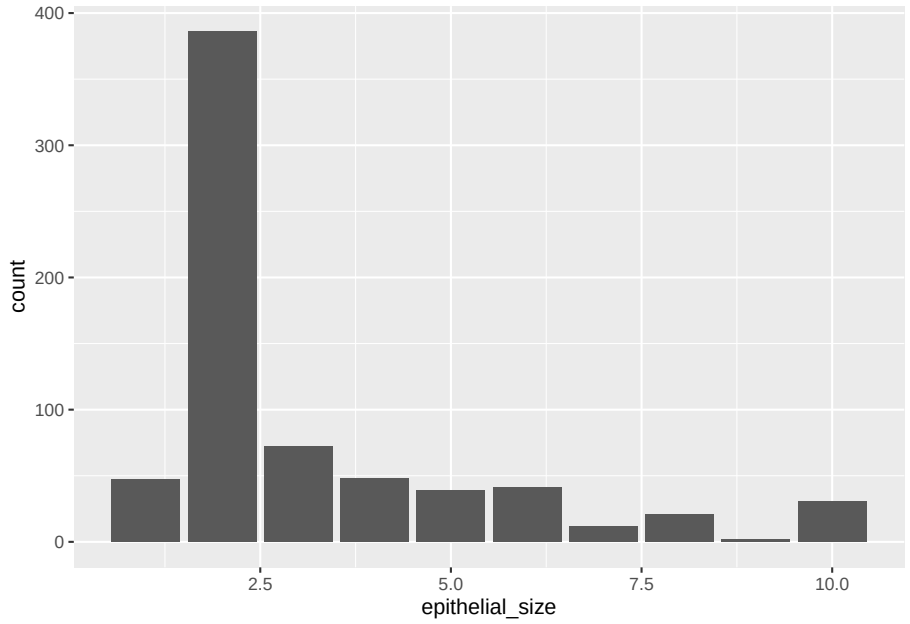
In any case, the x axis now shows the correct labels.

```
ggplot(df_cancer_new) +
  geom_bar(aes(x = class))
```



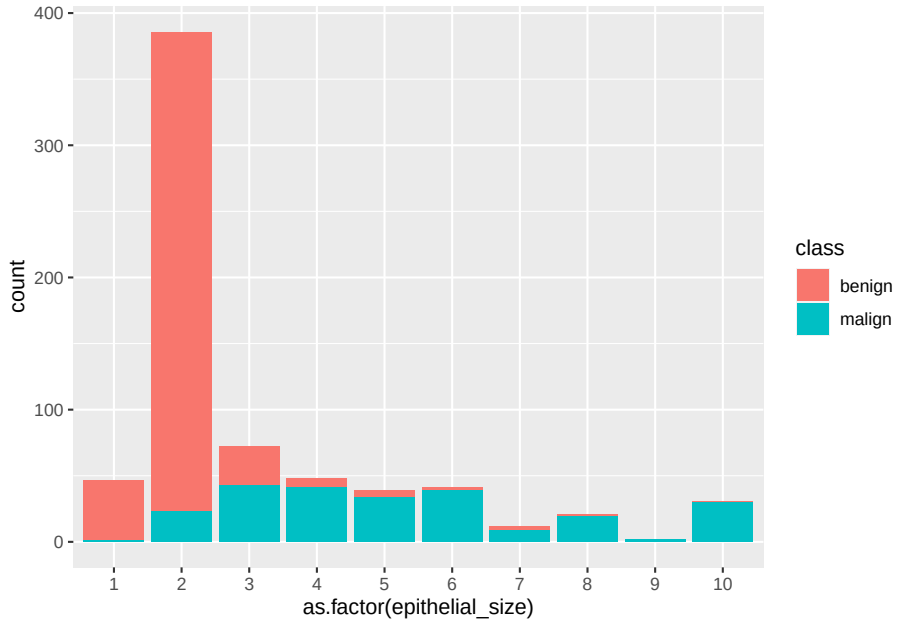
Let's mix some info. Here we plot the number of cases (the distribution) of `epithelial_size`.

```
ggplot(df_cancer_new) +
  geom_bar(aes(x = epithelial_size))
```



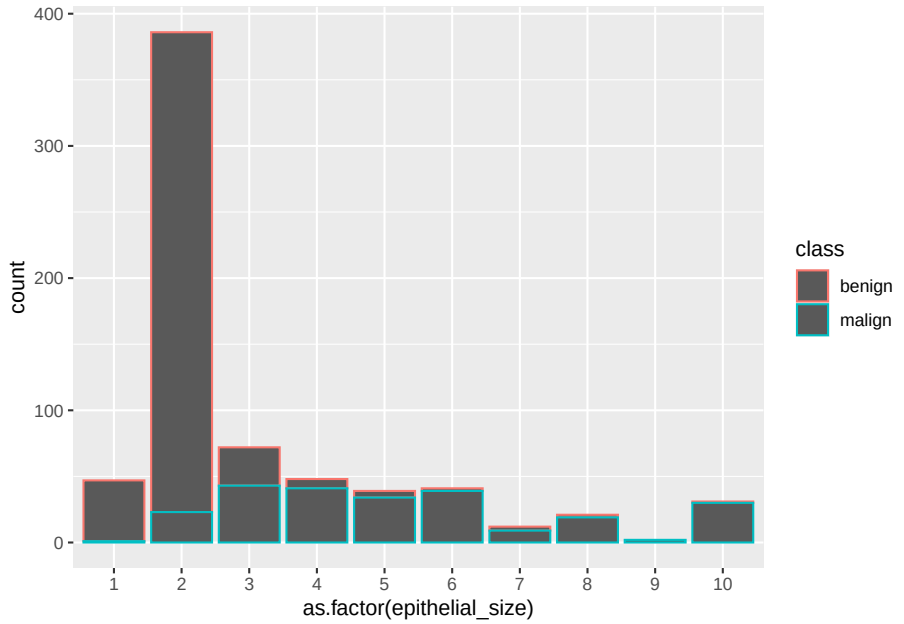
But we can tell ggplot to include the number of cases for the target variable. We want it to fill part of the column in one color and the rest in other color, depending on the number of malign and benign cases.

```
ggplot(df_cancer_new) +
  geom_bar(aes(x = as.factor(epithelial_size), fill = class))
```



Note that we used `fill=`. `col=` works differently.

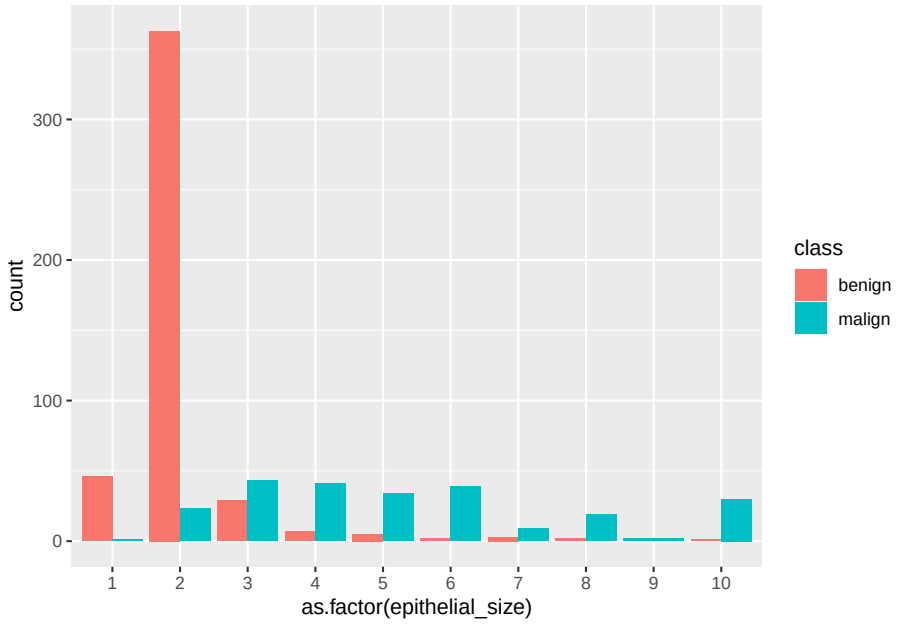
```
ggplot(df_cancer_new) +
  geom_bar(aes(x = as.factor(epithelial_size), col = class))
```



Take also into account that we are writing this statements (`fill=`) inside the `aes()` function, since we want to use information from the data frame for colouring. If we wanted to change the color of all the bars uniformly, we would write outside the `aes()` function something like `fill = "darkblue"`.

By default, ggplot plots the columns stacked, one on top of the other for the same class indicated in the `x=` atribute. Having them at the same level is called *dodge* and this can be achieved with the `position=` atribute. It doesn't depend on the data frame, so we write it outside the `aes()` function. By default, ggplot takes `position = "dodge"`. We can change it:

```
ggplot(df_cancer_new) +
  geom_bar(aes(x = as.factor(epithelial_size), fill = class), position = "dodge")
```

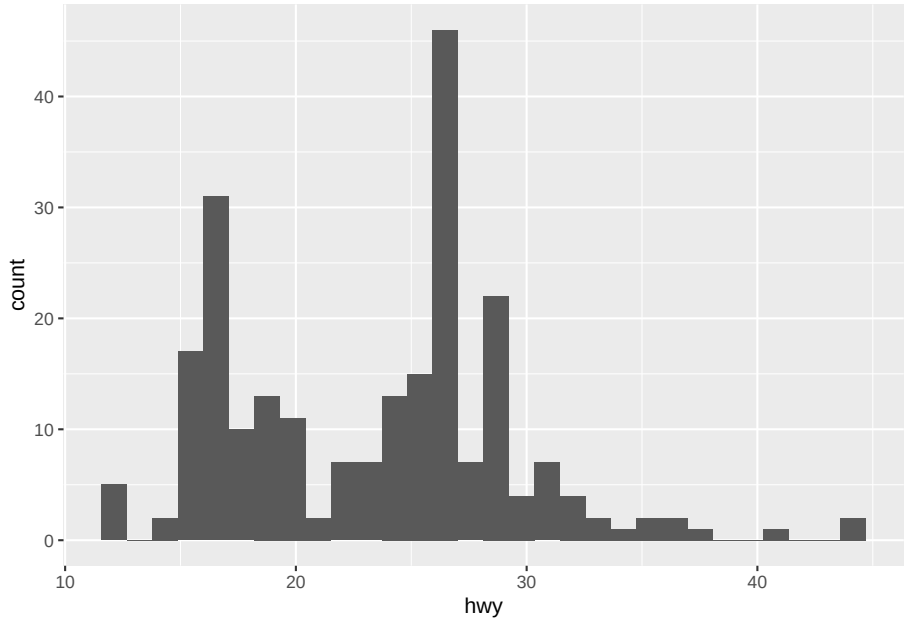


4.7 Histograms

Histograms are one of the most common plots for visualizing the distribution of one continuous variable (income, age, height, population,...). The `geom` used is `geom_histogram()` and the main parameter for the `aes()` function is `x=`, though you can also include some of the parameters available for `geom_bar()`. Have a look at the documentation for histograms.

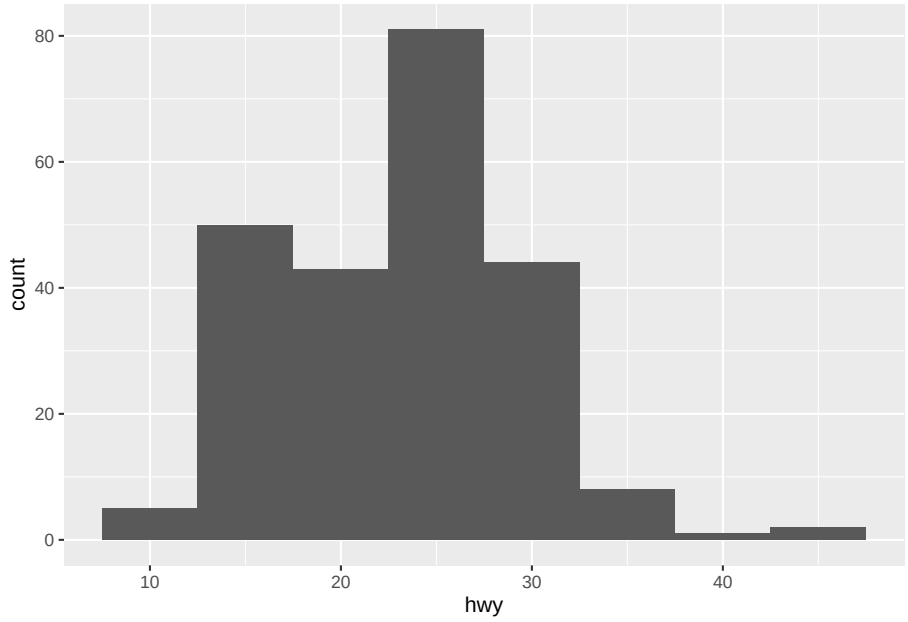
```
ggplot(mpg) +
  geom_histogram(aes(x = hwy))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



When plotting a histogram, we'll receive a message saying the number of bins (columns) used by the default. For avoiding it, we need to set a number of bins or, more appropriate, their width (in relation to the variable whose distribution we are studying).

```
ggplot(mpg) +
  geom_histogram(aes(x = hwy), binwidth = 5)
```



The plot changes a lot depending on this parameter, so be careful: a histogram with too many bins can be useless because it may not show differences among the values, but with too few can hide nuances.

Histograms are useful for getting an idea of the general distribution of a variable but not for specific values.

```
library(readr)
library(janitor)

df_countries <- read_csv("data/countries_of_the_world.csv", locale = locale(decimal_ma
  clean_names() %>%
  tidyr::drop_na()
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   .default = col_double(),
##   Country = col_character(),
##   Region = col_character()
## )
## i Use `spec()` for the full column specifications.
```

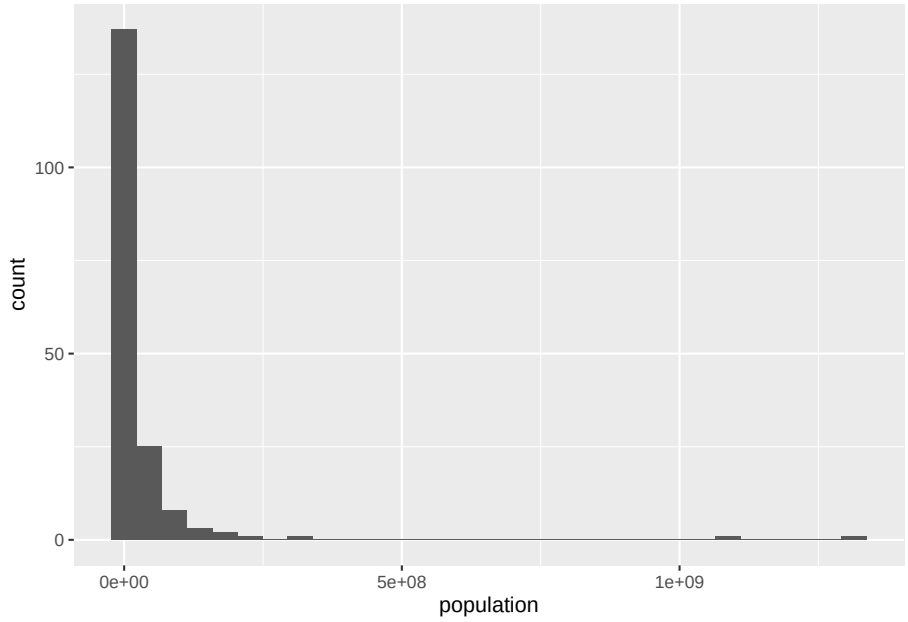
```
glimpse(df_countries)
```

```
## Rows: 179
## Columns: 20
## $ country                           <chr> "Afghanistan", "Albania", "Algeria", ~
## $ region                            <chr> "ASIA (EX. NEAR EAST)", "EASTERN EURO~
## $ population                        <dbl> 31056997, 3581655, 32930091, 13477, 6~
## $ area_sq_mi                        <dbl> 647500, 28748, 2381740, 102, 443, 276~
## $ pop_density_per_sq_mi             <dbl> 48.0, 124.6, 13.8, 132.1, 156.0, 14.4~
## $ coastline_coast_area_ratio        <dbl> 0.00, 1.26, 0.04, 59.80, 34.54, 0.18,~
## $ net_migration                     <dbl> 23.06, -4.93, -0.39, 10.76, -6.15, 0.~
## $ infant_mortality_per_1000_births <dbl> 163.07, 21.52, 31.00, 21.03, 19.46, 1~
## $ gdp_per_capita                    <dbl> 700, 4500, 6000, 8600, 11000, 11200, ~
## $ literacy_percent                  <dbl> 36.0, 86.5, 70.0, 95.0, 89.0, 97.1, 9~
## $ phones_per_1000                   <dbl> 3.2, 71.2, 78.1, 460.0, 549.9, 220.4,~
## $ arable_percent                    <dbl> 12.13, 21.09, 3.22, 0.00, 18.18, 12.3~
## $ crops_percent                     <dbl> 0.22, 4.42, 0.25, 0.00, 4.55, 0.48, 2~
## $ other_percent                     <dbl> 87.65, 74.49, 96.53, 100.00, 77.27, 8~
## $ climate                           <dbl> 1.0, 3.0, 1.0, 2.0, 2.0, 3.0, 4.0, 2.~
## $ birthrate                         <dbl> 46.60, 15.11, 17.14, 14.17, 16.93, 16~
## $ deathrate                         <dbl> 20.34, 5.22, 4.61, 5.34, 5.37, 7.55, ~
## $ agriculture                       <dbl> 0.380, 0.232, 0.101, 0.040, 0.038, 0.~
## $ industry                          <dbl> 0.240, 0.188, 0.600, 0.180, 0.220, 0.~
## $ service                           <dbl> 0.380, 0.579, 0.298, 0.780, 0.743, 0.~
```

If we don't control a bit the values, the plot won't be useful. For instance, histograms are very sensitive to extreme values.

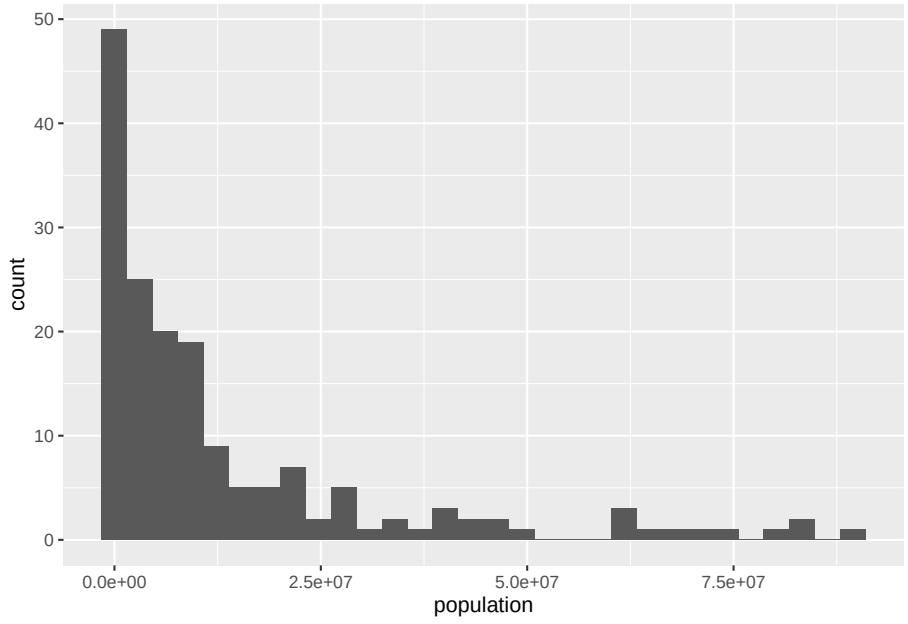
```
ggplot(df_countries) +
  geom_histogram(aes(x = population))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



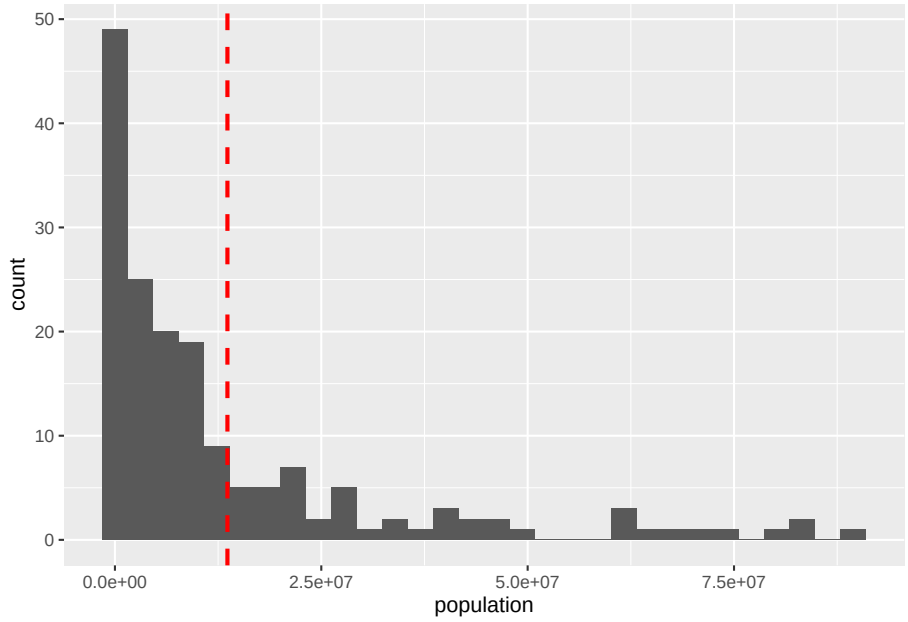

```
df_countries_redux <- df_countries %>%
  # filter(!country %in% c("China", "India")) %>%
  filter(population <= 100000000)

ggplot(df_countries_redux) +
  geom_histogram(aes(x = population))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



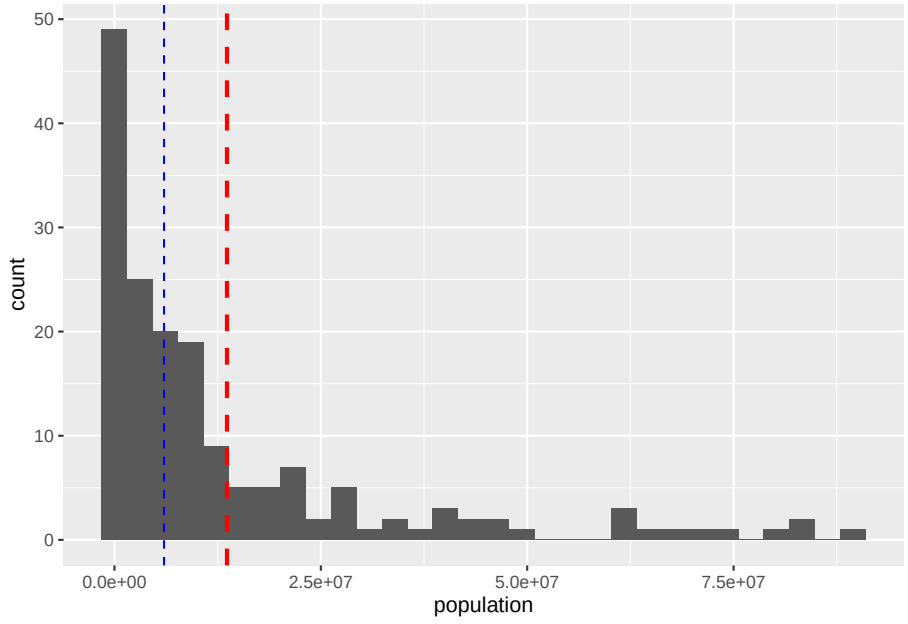
From the plot, could you say what is the mean of the population?

```
df_media <- df_countries_redux %>%
  summarise(media = mean(population),
            mediana = median(population),
            q1 = quantile(population, probs = 0.25),
            q3 = quantile(population, probs = 0.75))

ggplot(df_countries_redux) +
  geom_histogram(aes(x = population), bins = 30) +
  geom_vline(xintercept = df_media$media, col = "red", size = 1, linetype = 2)
```



Would you say than the median is higher or lower?

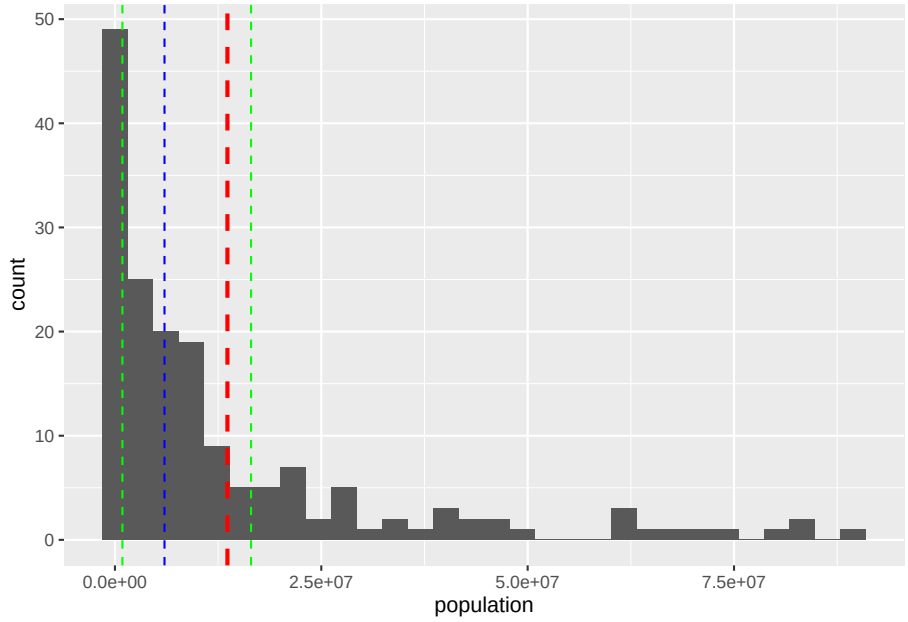
```
ggplot(df_countries_redux) +
  geom_histogram(aes(x = population), bins = 30) +
  geom_vline(xintercept = df_media$media, col = "red", size = 1, linetype = 2) +
  geom_vline(xintercept = df_media$mediana, col = "blue", size = 0.5, linetype = 2)
```



The mean is very sensitive to outliers. The median is usually a better aggregation measure when the distribution is not symmetrical.

We can also include the quartiles.

```
ggplot(df_countries_redux) +
  geom_histogram(aes(x = population), bins = 30) +
  geom_vline(xintercept = df_media$media, col = "red", size = 1, linetype = 2) +
  geom_vline(xintercept = df_media$mediana, col = "blue", size = 0.5, linetype = 2) +
  geom_vline(xintercept = df_media$q1, col = "green", size = 0.5, linetype = 2) +
  geom_vline(xintercept = df_media$q3, col = "green", size = 0.5, linetype = 2)
```

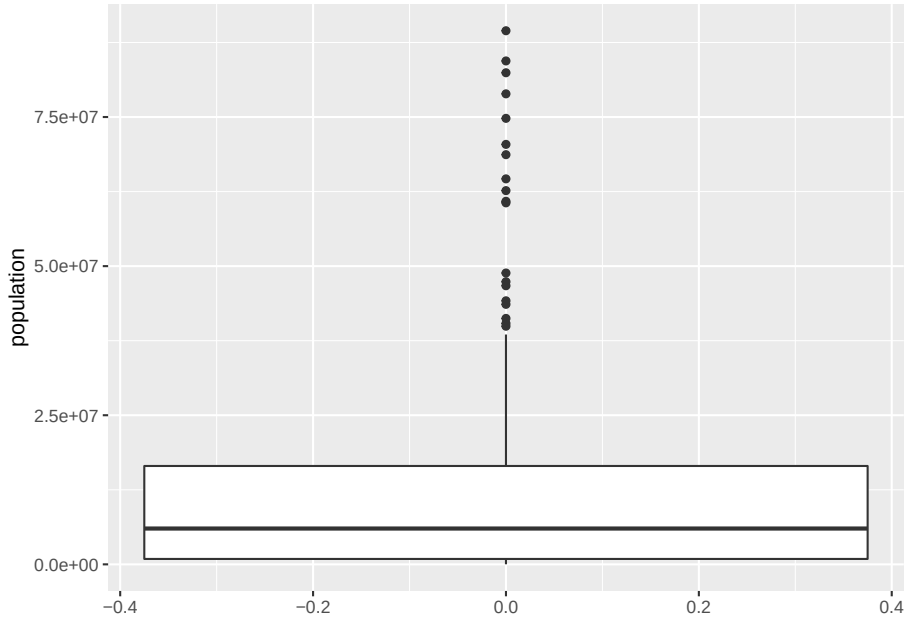


Between the two green lines there are 50% of the countries (after excluding the largest ones).

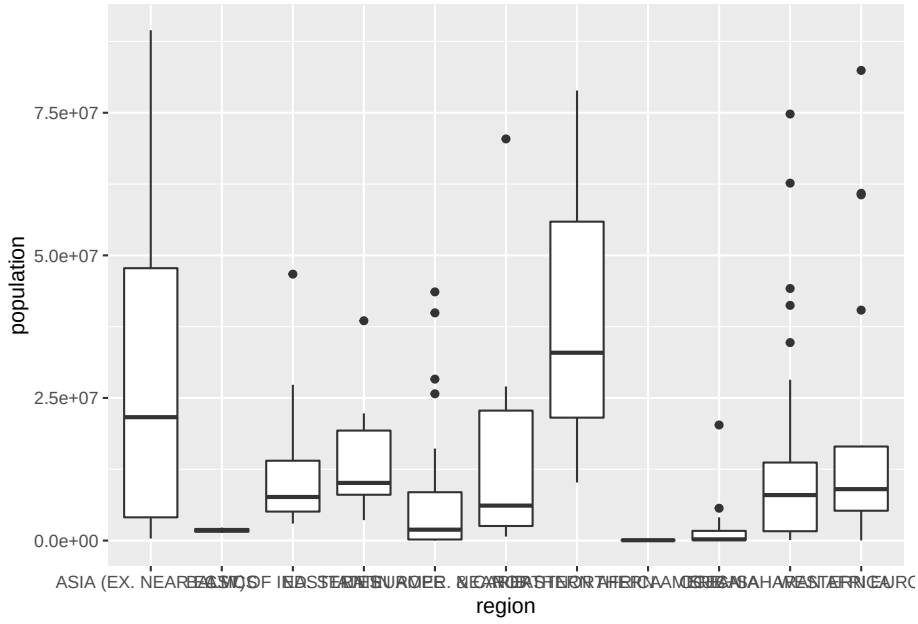
So, the histogram allows us seeing some general properties of the distribution (outliers, asymmetrical distribution, lots of observations concentrated around a particular value) but it's not comfortable for detecting some typical statistics.

4.8 Boxplots

```
ggplot(df_countries_redux) +
  geom_boxplot(aes(y = population))
```



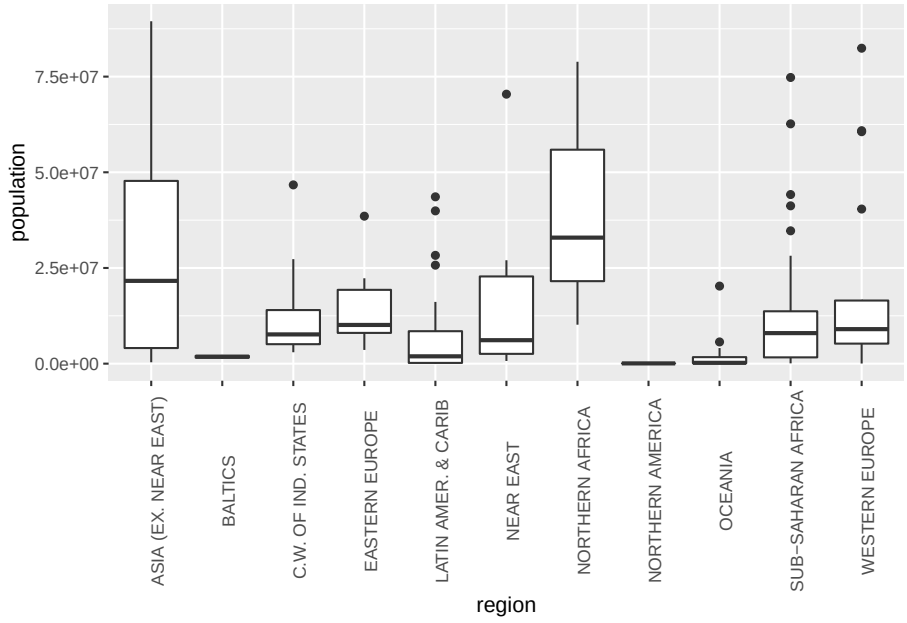

```
ggplot(df_countries_redux) +
  geom_boxplot(aes(x = region, y = population))
```



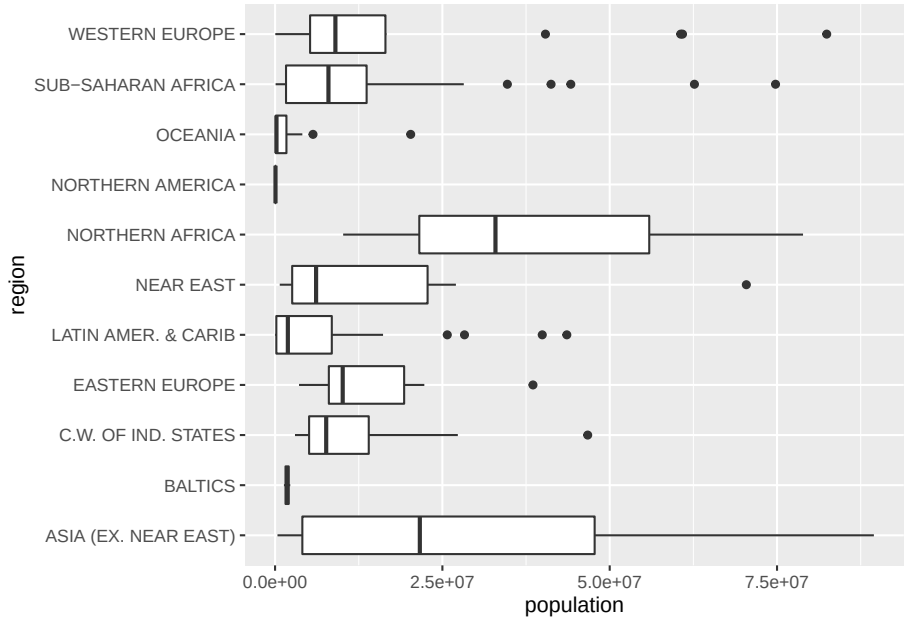
4.9 Some improvements

Not the best option here, but useful sometimes:

```
ggplot(df_countries_redux) +
  geom_boxplot(aes(x = region, y = population)) +
  theme(axis.text.x = element_text(angle = 90))
```

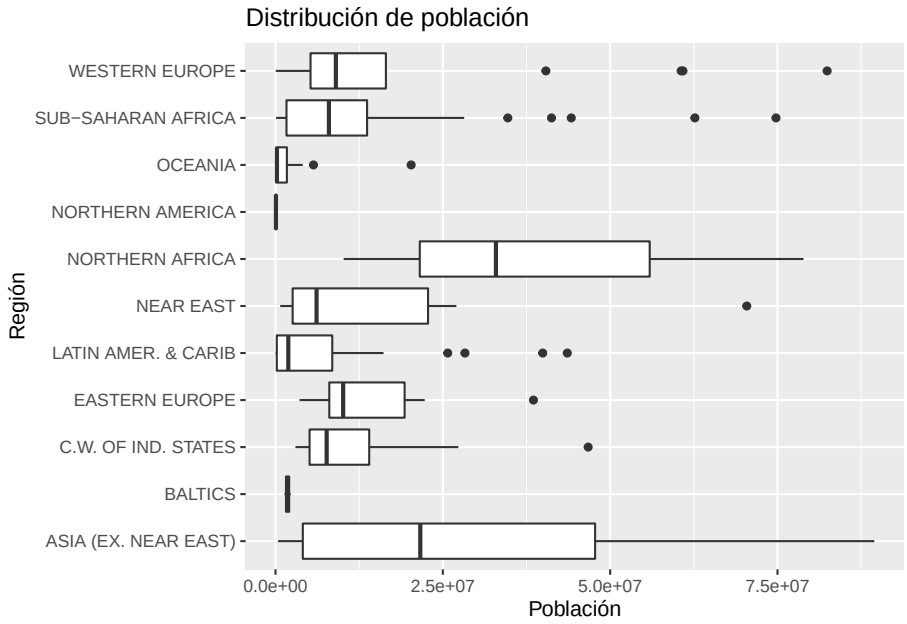



```
ggplot(df_countries_redux) +
  geom_boxplot(aes(x = region, y = population)) +
  coord_flip()
```



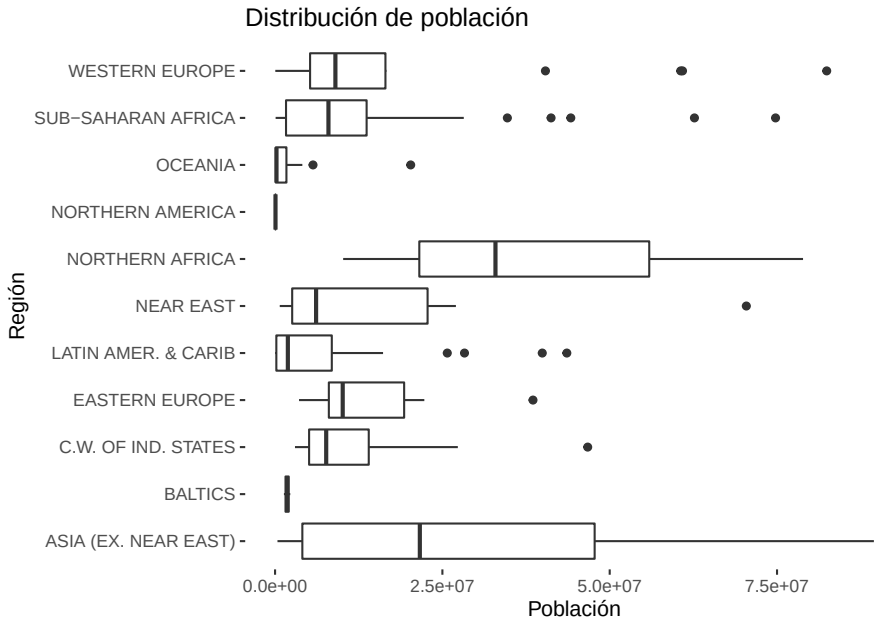
Some changes in the labels:

```
ggplot(df_countries_redux) +
  geom_boxplot(aes(x = region, y = population)) +
  labs(x = "Región", y = "Población", title = "Distribución de población") +
  coord_flip()
```



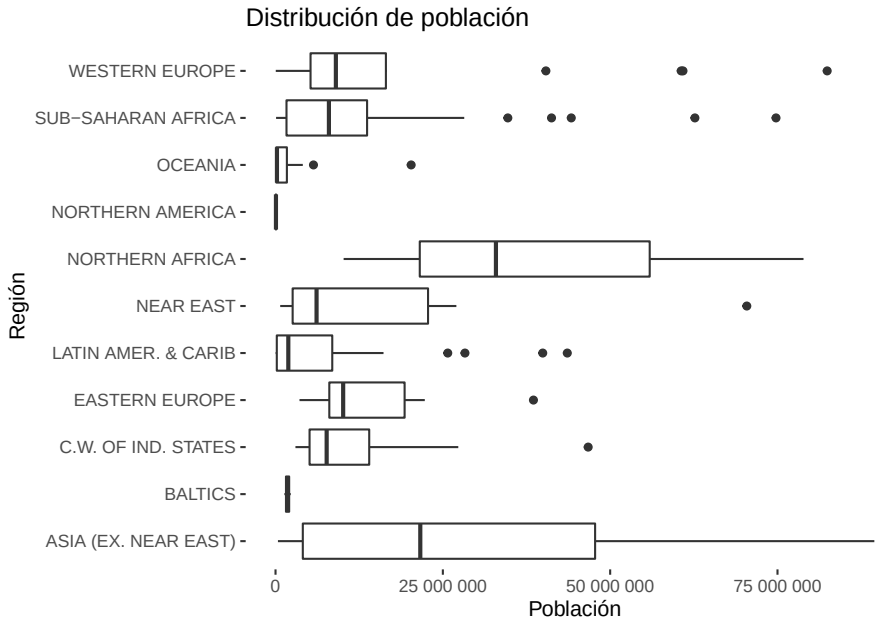
We can also set a white rectangle, instead of grey.

```
ggplot(df_countries_redux) +
  geom_boxplot(aes(x = region, y = population)) +
  labs(x = "Región", y = "Población", title = "Distribución de población") +
  coord_flip() +
  theme(panel.background = element_blank())
```



Changing the format of the numbers is a bit tricky (espcially for the Spanish format: we show here the English format).

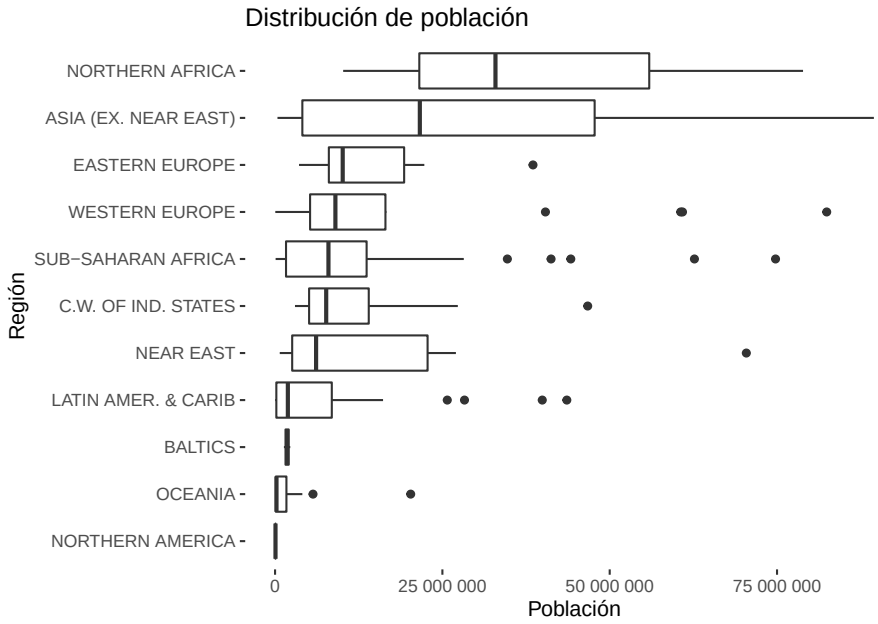
```
ggplot(df_countries_redux) +
  geom_boxplot(aes(x = region, y = population)) +
  labs(x = "Región", y = "Población", title = "Distribución de población") +
  coord_flip() +
  scale_y_continuous(labels = scales::number) +
  theme(panel.background = element_blank())
```



Data visualization good practices recommend arranging the data. In some cases this can be achieved easily with ggplot, but in this boxplot case we will need dplyr and the factor class.

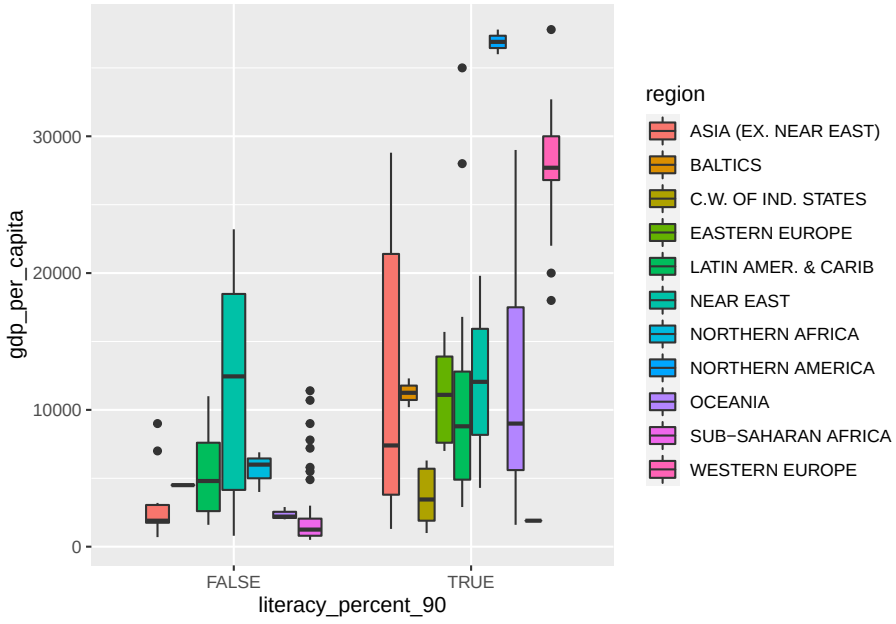
There are several approaches: this is just one.

```
df_regiones_ordenadas <- df_countries_redux %>%
  group_by(region) %>%
  summarise(mediana = median(population)) %>%
  arrange(mediana) %>%
  mutate(region = forcats::as_factor(region))


df_countries_redux %>%
  mutate(region = factor(region, levels = df_regiones_ordenadas$region)) %>%
  ggplot() +
  geom_boxplot(aes(x = region, y = population)) +
  labs(x = "Región", y = "Población", title = "Distribución de población") +
  coord_flip() +
  scale_y_continuous(labels = scales::number) +
  theme(panel.background = element_blank())
```



4.10 Mixing data

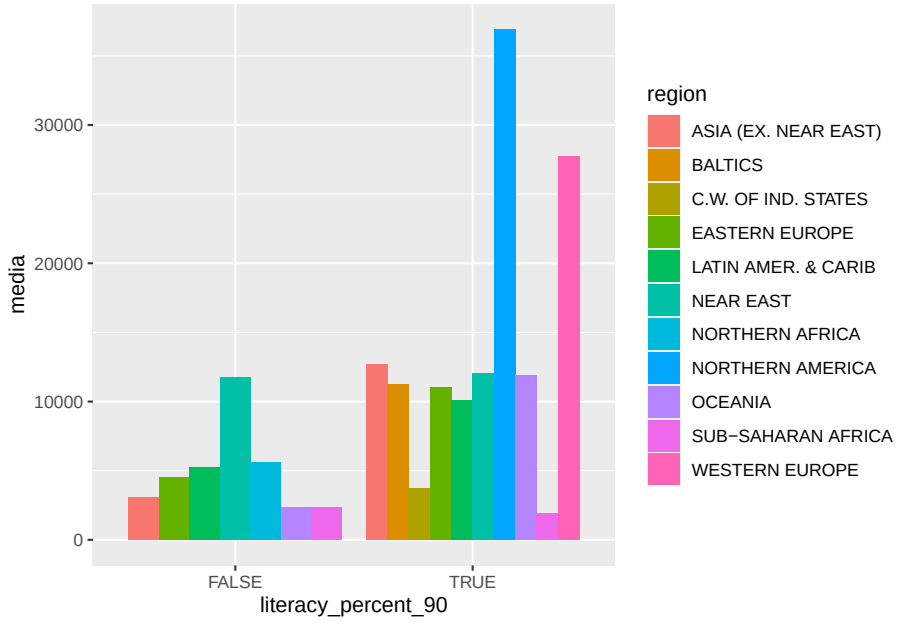
```
df_countries %>%
  mutate(literacy_percent_90 = literacy_percent > 90) %>%
  ggplot() +
  geom_boxplot(aes(x = literacy_percent_90, y = gdp_per_capita, fill = region))
```



4.10.1 Exercise

Generate a column plot similar to the previous boxplot, but comparing the average of the gpd_per_capita. *Hint.* Have a look at the documentation of `geom_col()` for learning how to set a dodge position in the columns.

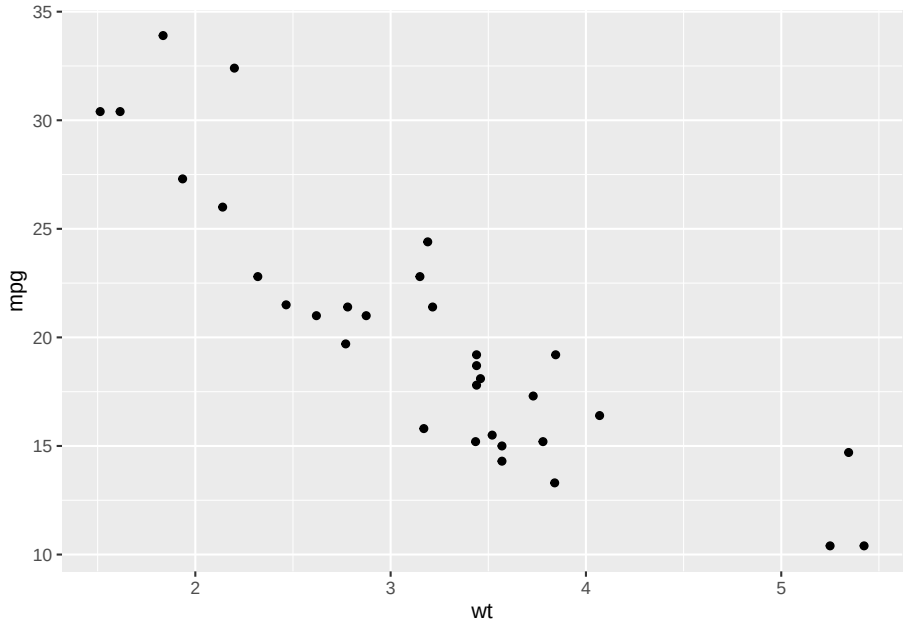
```
## `summarise()` has grouped output by 'region'. You can override using the `.groups` a
```



4.11 Facets: The basics

Firstly, we create a scatter plot comparing the miles per gallon variable against the weight of the car, from the `mtcars` data frame.

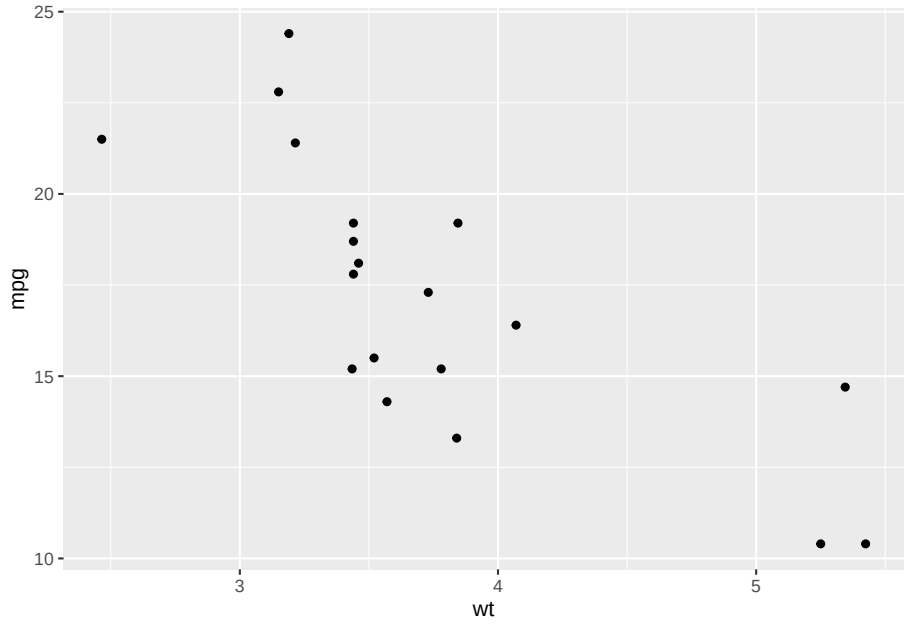
```
library(ggplot2)

ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point()
```



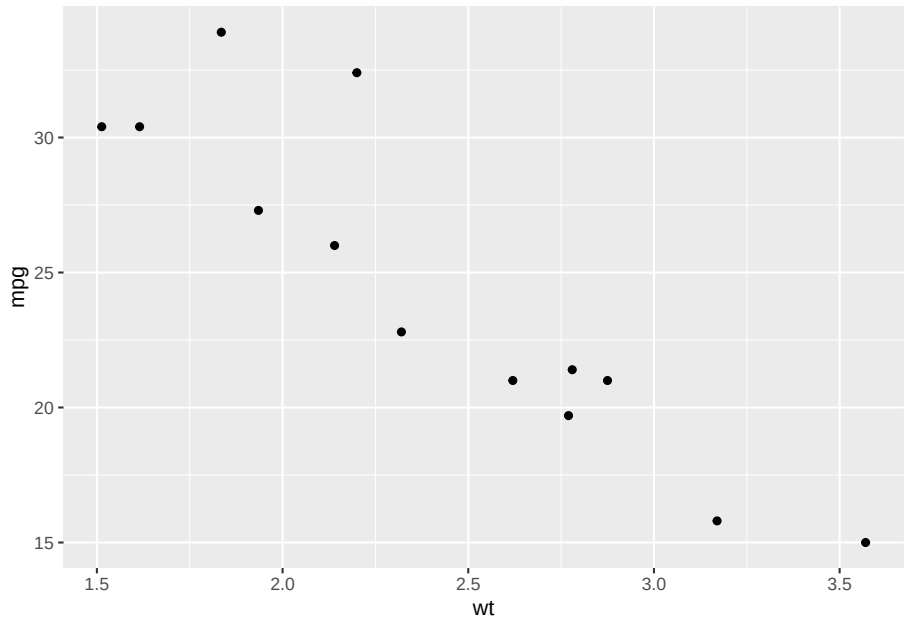
We know want to see what happens when we take into account the categorical variable Transmission type `am`. We could try plotting the points with different colors or shapes depending on their `am` value but, when possible, the best way is comparing their position on different plots that have the same axis.

We would like to do something like these two plots (we combine dplyr and ggplot2):

```
library(dplyr)

mtcars %>%
  filter(am == 0) %>%
  ggplot(aes(x = wt, y = mpg)) +
  geom_point()
```



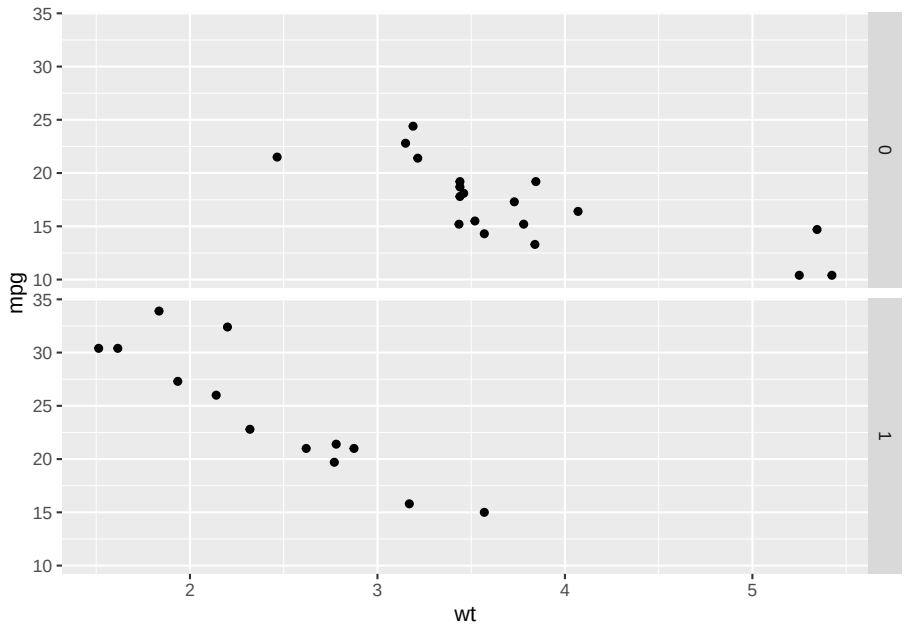

```
mtcars %>%
  filter(am == 1) %>%
  ggplot(aes(x = wt, y = mpg)) +
  geom_point()
```



But they don't have the same axis and are not disposed in a proper way, so comparing them for understanding the differences between the two distributions is not comfortable. ggplot2 may help.

We are going to create two scatter plots: one per each value of the `am` column. Both they'll have the same axis, so it'll be easier comparing the relative position of the dots, which is the easiest way of getting differences between the values of a continuous variable.

```
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  facet_grid(am ~ .)
```



In the plot is easy to see that automatic cars (`am == 0`) heavier (x axis) and that manual cars are associated with more miles per gallon (y axis).

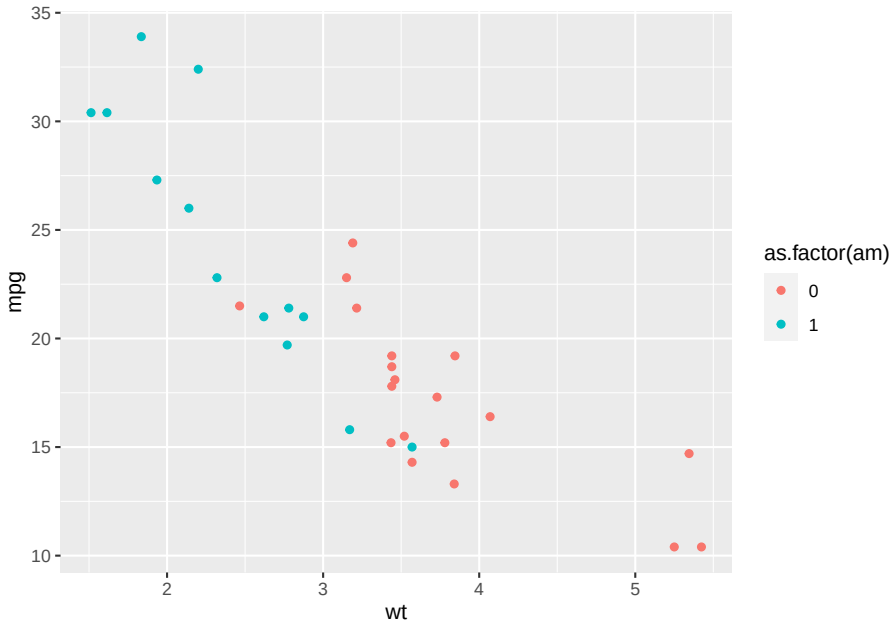
Let's have a look at the sintax. The `ggplot()` and the `geom_point()` have already been study. What we are doing is adding a new layer, a faceting layer. What we achieve is that we will repeat the plot generated with the `ggplot() + geom_point()` schema, for each value in the `am` column. The `facet_grid()` function creates a grid for all the desired plot. It receives a formula based on the `~` sintax. Before the `~` we include the variable whose values will define the rows of the grid; after the `~`, we write the variable for defining the columns. A `.` is written when no difference is made.

Remember. For comparing a continuous variable for several observations,

focusing on the position is more comfortable than changing the colors or the shape.

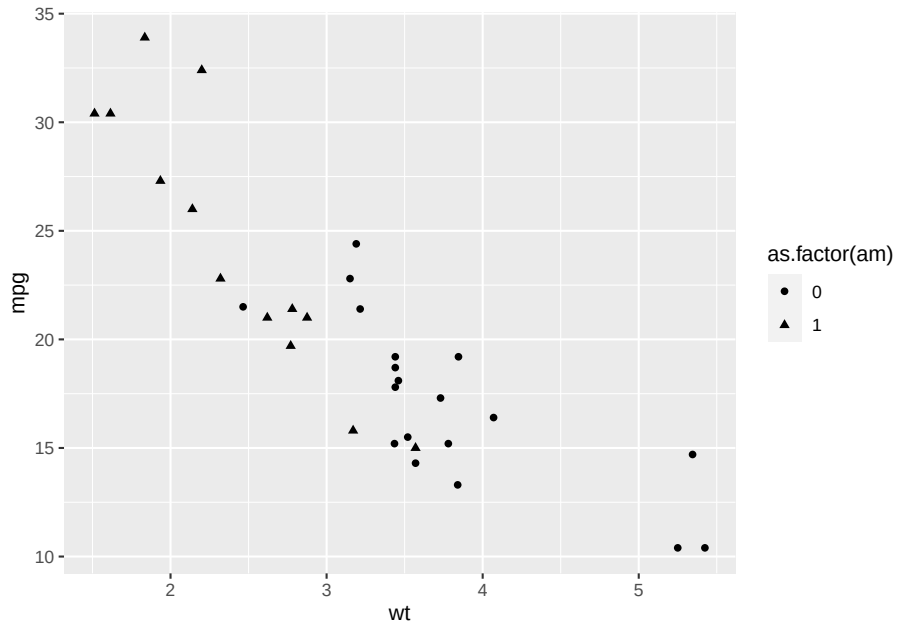
Let's try the same comparison with different colors for each point, depending on their transmission.

```
ggplot(mtcars, aes(x = wt, y = mpg, color = as.factor(am))) +
  geom_point()
```



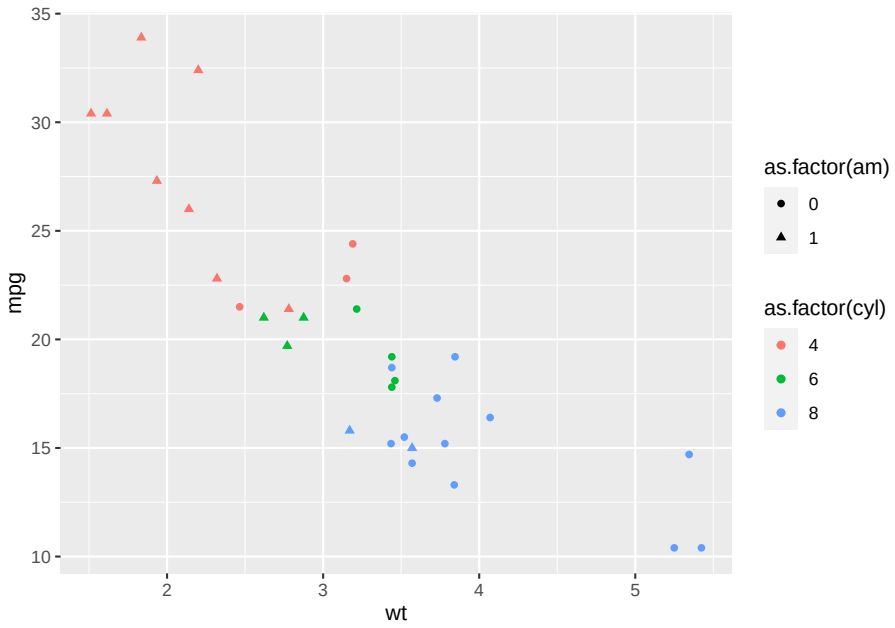
Or the shape.

```
ggplot(mtcars, aes(x = wt, y = mpg, shape = as.factor(am))) +
  geom_point()
```



Now we want to include a new variable: the number of cylinders. Let's assume that we preferred the previous plot with the shape aesthetic. Let's add the color for understanding differences between the `cyl` variable.

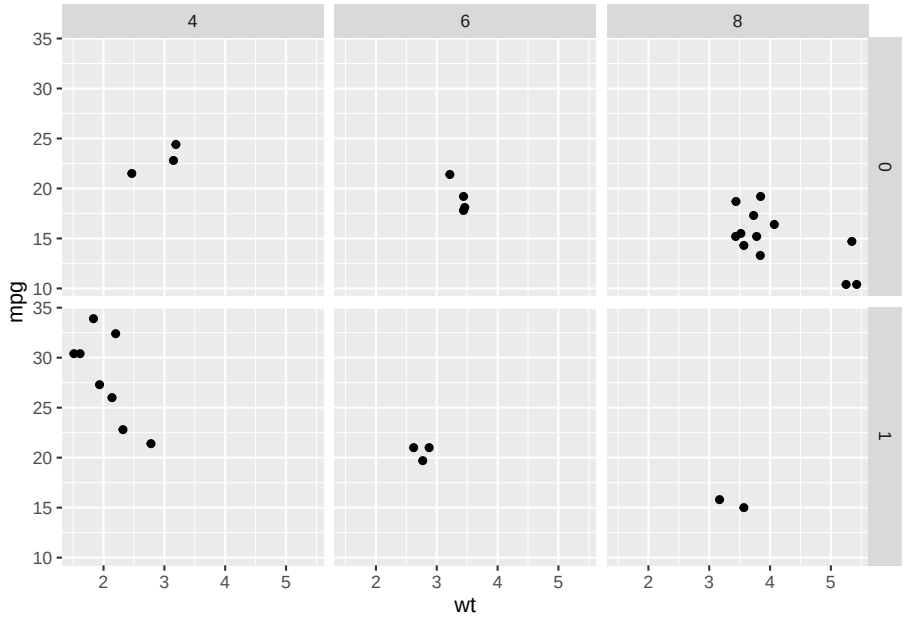
```
ggplot(mtcars, aes(x = wt, y = mpg, shape = as.factor(am), color = as.factor(cyl))) +
  geom_point()
```



A mess. Again, it is better taking advantage of the spatial distribution, since our eye understand better differences on the position.

For doing this, facets:

```
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  facet_grid(am ~ cyl)
```



Here, we define the grid from the `am` variable and the `cyl` variable for the columns.

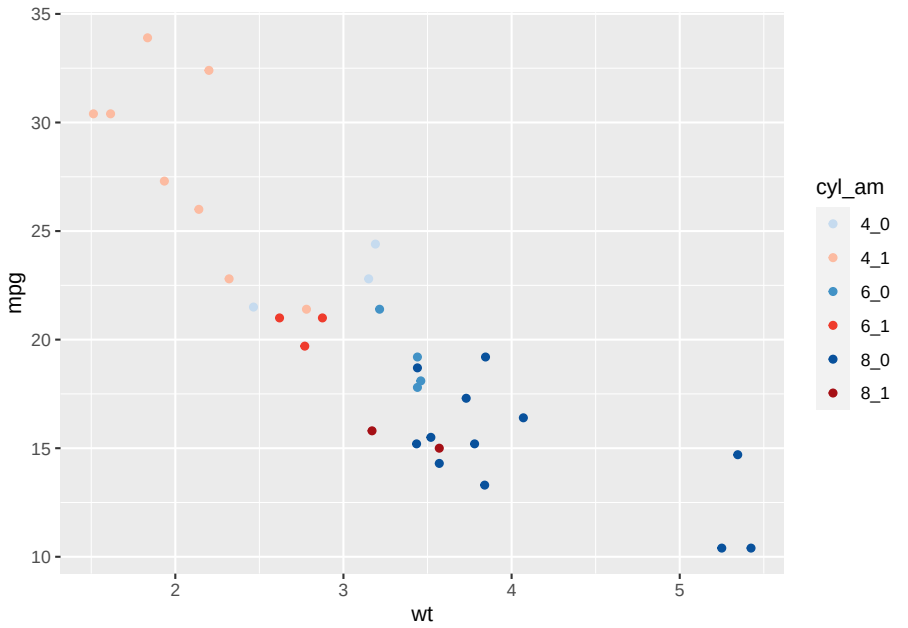
4.12 Many variables in just one plot

Know, an extreme example is shown. The next pieces of codes will help us creating a plot where up to 7 variables are studied.

First we create the set of color that we'll use for the points. We define a special object we haven't used yet: a matrix. It will contain character elements with the desired colors (instead of names of color, hexadecimal codes are used). The blue colors will be associated with the automatic cars and the red colors, with the manual cars. Lighter colors will be related to lower number of cylinders (for this transparency, the `brewer.pal()` will take care of it).

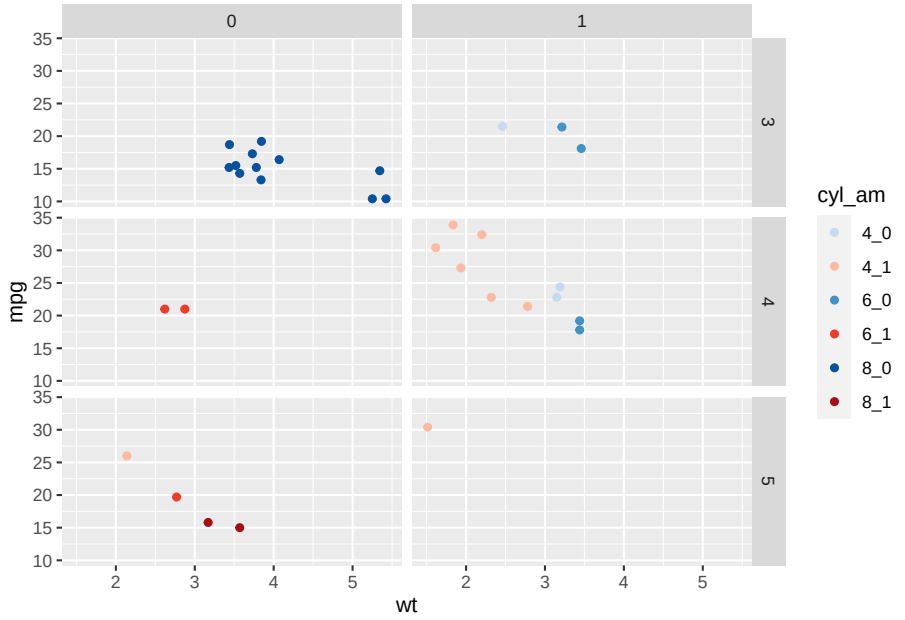
```
mtcars <- mtcars %>%
  mutate(cyl_am = paste(cyl, am, sep = "_"))

myCol <- rbind(RColorBrewer::brewer.pal(9, "Blues")[c(3,6,8)],
               RColorBrewer::brewer.pal(9, "Reds")[c(3,6,8)])

# Basic scatter plot, add color scale:
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl_am)) +
```

```
  geom_point()+
  scale_color_manual(values = myCol)
```



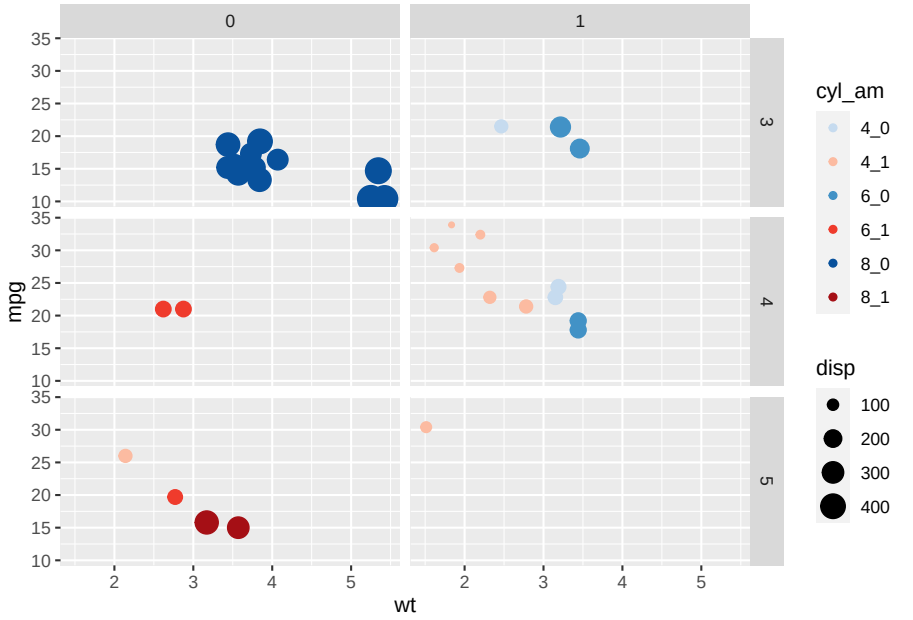
Now we also include a visualization for getting differences considering `gear` and `vs`. This can be done with facets.

```
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl_am)) +
  geom_point()+
  scale_color_manual(values = myCol)+
  facet_grid(gear ~ vs)
```



Finally, we include information about `disp`, mapping it to the size of the points.

```
ggplot(mtcars, aes(x = wt, y = mpg, col = cyl_am, size = disp)) +
  geom_point() +
  scale_color_manual(values = myCol) +
  facet_grid(gear~vs)
```



4.13 Something more

4.13.1 Pivoting data

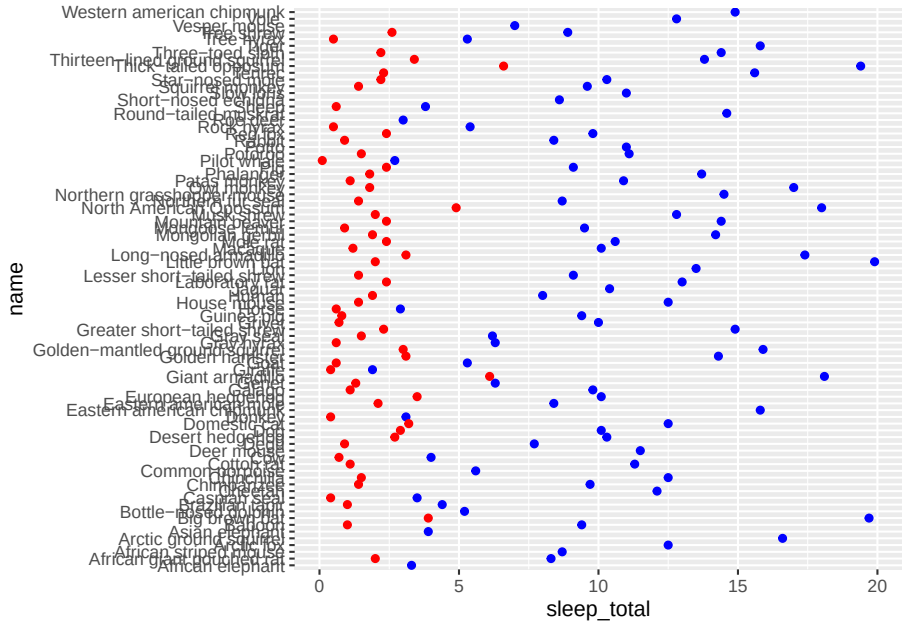
Data will often be in such a format that cannot be properly used for visualization. The packages on the *tidyverse* environment can help us dealing with this.

On the `msleep` dataset we have information about how much time some mammals sleep. About this time, there are two columns: `sleep_total` and `sleep_rem`. We want to visualize both times depending on the animal. How can we visualize this two variables at the same time, if they are two columns?

A first approach could be using two layers of points.

```
ggplot(msleep) +
  geom_point(aes(x = sleep_total, y = name), color = "blue") +
  geom_point(aes(x = sleep_rem, y = name), color = "red")
```

```
## Warning: Removed 22 rows containing missing values (geom_point).
```



Things we don't like about this plot:

- There is no legend (it can be added but with more code).
- We had to decide the colors explicitally (ggplot2 usually knows how to decide what color is the best).
- If we would like to work with more categories, we would have to add one more like per category (and decide a color).
- The x axis has the label of the first layer. It can be changed but, again, is one more line.

Instead of this, we would like to specify ggplot that it has to create a plot of the name of the mammal against the sleeping time. This *time* unit is divided into several categories (rem and total), and we will use these categories for coloring the points. To sum up, right now in the data frame we have one row per mammal, and two columns about the sleeping time. We want to redefine the data frame so that we will have two rows per mammal: one for the total time and one for the rem time.

This is called pivoting. Let's have a look at the actual data (we select only the columns that will be used):

```
msleep %>%
  select(vore, name, sleep_total, sleep_rem) %>%
  slice(1:6)
```

```
## # A tibble: 6 x 4
##   vore  name                        sleep_total sleep_rem
##   <chr> <chr>                             <dbl>     <dbl>
## 1 carni Cheetah                            12.1      NA
## 2 omni  Owl monkey                         17         1.8
## 3 herbi Mountain beaver                    14.4       2.4
## 4 omni  Greater short-tailed shrew         14.9       2.3
## 5 herbi Cow                                 4         0.7
## 6 herbi Three-toed sloth                   14.4       2.2
```

What we want to achieve is this:

```
## # A tibble: 12 x 4
##    vore  name                        sleep        time
##    <chr> <chr>                       <chr>       <dbl>
##  1 carni Cheetah                     sleep_total  12.1
##  2 carni Cheetah                     sleep_rem    NA
##  3 omni  Owl monkey                  sleep_total  17
##  4 omni  Owl monkey                  sleep_rem     1.8
##  5 herbi Mountain beaver             sleep_total  14.4
##  6 herbi Mountain beaver             sleep_rem     2.4
##  7 omni  Greater short-tailed shrew  sleep_total  14.9
##  8 omni  Greater short-tailed shrew  sleep_rem     2.3
##  9 herbi Cow                         sleep_total   4
## 10 herbi Cow                         sleep_rem     0.7
## 11 herbi Three-toed sloth            sleep_total  14.4
## 12 herbi Three-toed sloth            sleep_rem     2.2
```

The information is the same on both tables, but the second one has a long format (more rows) and the first one has a wide format (more columns). Depending on what you are doing, you will prefer one or the other. For instance, for statistical modelling, the wide format may be preferred; for data visualization, the longer one is preferred (at least, for this example).

Changing a data frame between this two formats is called pivoting. In R, there lots of tools for achieving this. As usual, we show one:

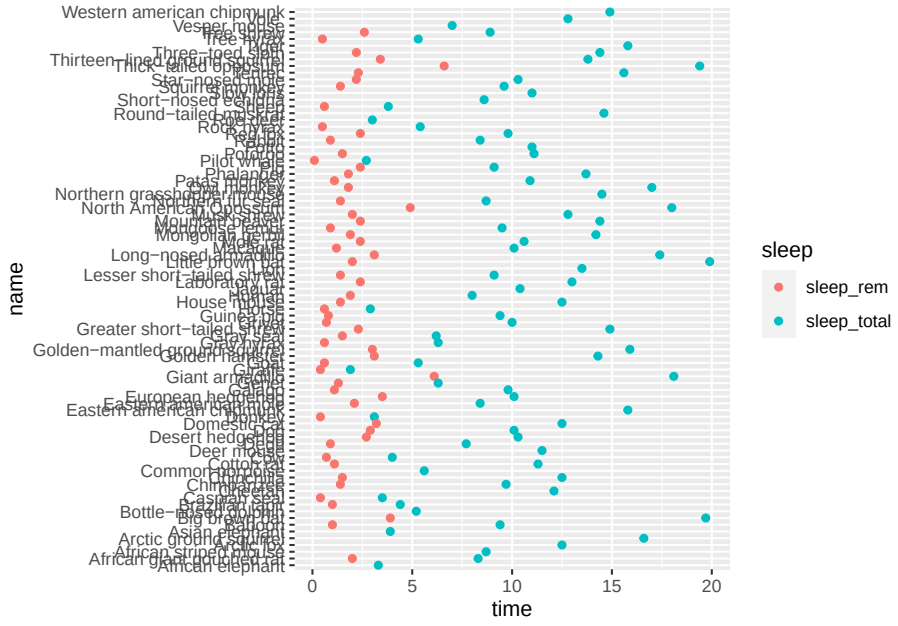
```
library(tidyr)

mamsleep <- msleep %>%
  select(vore, name, sleep_total, sleep_rem) %>%
  pivot_longer(cols = one_of(c("sleep_total", "sleep_rem")), names_to = "sleep", values_to = "tim

glimpse(mamsleep)
```

```
## Rows: 166
## Columns: 4
## $ vore  <chr> "carni", "carni", "omni", "omni", "herbi", "herbi", "omni", "omn~
## $ name  <chr> "Cheetah", "Cheetah", "Owl monkey", "Owl monkey", "Mountain beav~
## $ sleep <chr> "sleep_total", "sleep_rem", "sleep_total", "sleep_rem", "sleep_t~
## $ time  <dbl> 12.1, NA, 17.0, 1.8, 14.4, 2.4, 14.9, 2.3, 4.0, 0.7, 14.4, 2.2, ~
```

4.13.2 Basic scatter plot

```
ggplot(mamsleep,aes(x = time, y = name, col = sleep)) +
  geom_point()
```

```
## Warning: Removed 22 rows containing missing values (geom_point).
```

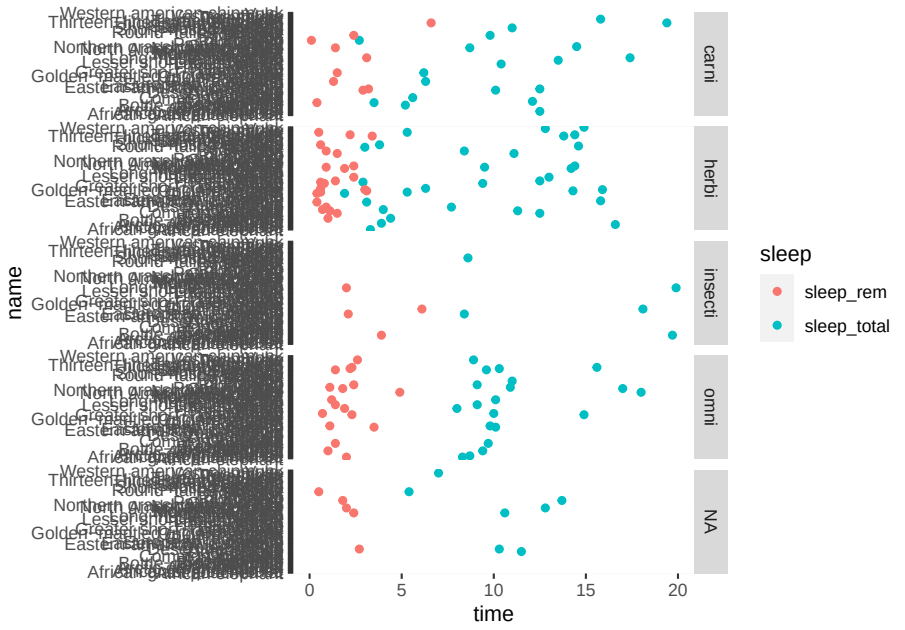


4.13.3 Facet rows accoding to vore

Now we separate the plot in several parts, depending on what the mammals eat. We will prepare a grid, as we did before, using facets.

```
ggplot(mamsleep, aes(x = time, y = name, col = sleep)) +
geom_point() +
facet_grid(vore ~ .)
```

```
## Warning: Removed 22 rows containing missing values (geom_point).
```



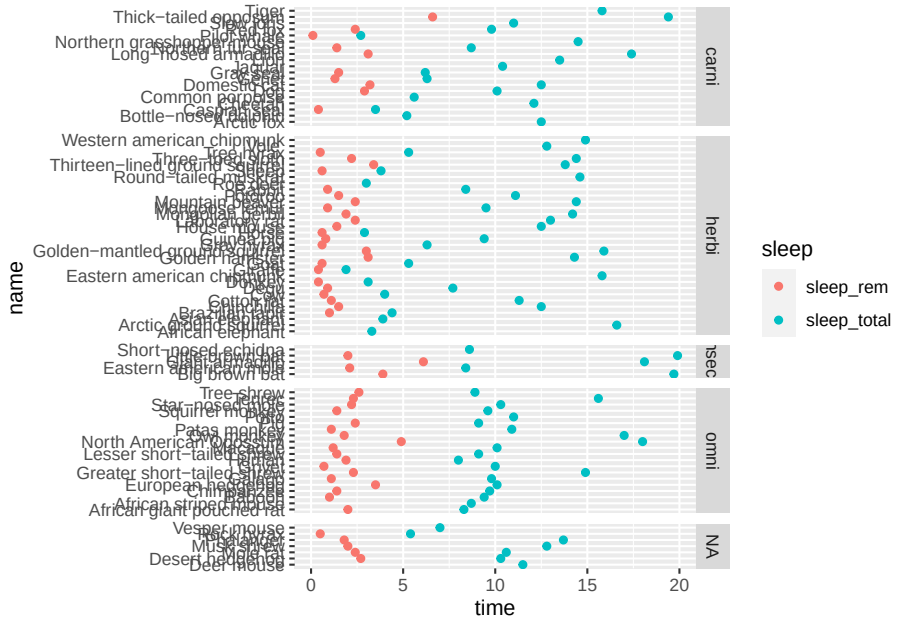
It is as if we had made for plots: a first one having `vore == "carni"`, a second one with `vore == "herbi"`, etc.

4.13.4 Specify scale and space arguments to free up rows

The x axis must be common for all the plots, because it is the most comfortable way for comparing the `time` value among the different points, even if they are on different parts of the grid. However, there is no need on having the same y axis on all of them, because it's not that what we are comparing. We can free the y axis, so that it shows only the important values for each plot.

```
ggplot(mamsleep, aes(x = time, y = name, col = sleep ))+
geom_point()+
facet_grid(vore ~ ., scales = "free_y", space = "free_y")
```

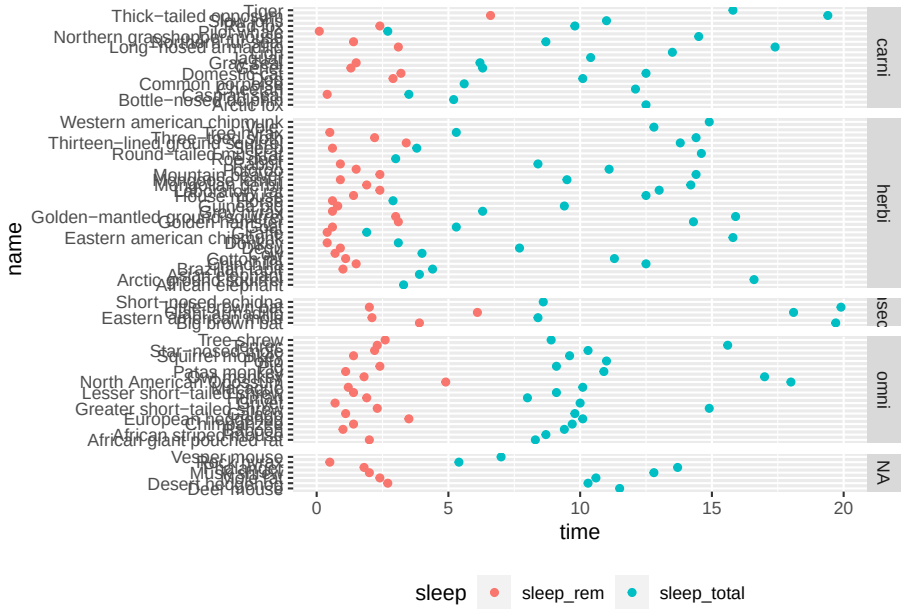
```
## Warning: Removed 22 rows containing missing values (geom_point).
```



For gaining some space, we can set the legend on the bottom of the plot.

```
ggplot(mamsleep, aes(x = time, y = name, col = sleep )) +
  geom_point() +
  facet_grid(vore ~ ., scales = "free_y", space = "free_y") +
  theme(legend.position = "bottom")
```

```
## Warning: Removed 22 rows containing missing values (geom_point).
```



This exercise, except the last section, has been downloaded from Datacamp and it is a project created by Antonio Sánchez Chinchón, data scientist at Telefonica.

4.14 United Nations life expectancy data

Life expectancy at birth is a measure of the average a living being is expected to live. It takes into account several demographic factors like gender, country, or year of birth.

Life expectancy at birth can vary along time or between countries because of many causes: the evolution of medicine, the degree of development of countries, or the effect of armed conflicts. Life expectancy varies between gender, as well. The data shows that women live longer that men. Why? Several potential factors, including biological reasons and the theory that women tend to be more health conscious.

Let's create some plots to explore the inequalities about life expectancy at birth around the world. We will use a dataset from the United Nations Statistics Division, which is available here.

For doing this:

- Load the readr, dplyr, tidyr and ggplot2 packages.
- Read `UNdata.csv` into a data frame and name it `life_expectancy`.
- Print the first few rows of life_expectancy.

- Have a look at the first rows with `head()`.

```
# This sets plot images to a nice size
options(repr.plot.width = 6, repr.plot.height = 6)

# Loading packages
library(...)
library(...)
library(...)
library(...)

# Loading data
... <- read_csv(...)

# Taking a look at the first few rows
head(life_expectancy)
```

```
##   Country.or.Area Subgroup      Year
## 1     Afghanistan   Female 2000-2005
## 2     Afghanistan   Female 1995-2000
## 3     Afghanistan   Female 1990-1995
## 4     Afghanistan   Female 1985-1990
## 5     Afghanistan     Male 2000-2005
## 6     Afghanistan     Male 1995-2000
##                                                                 Source  Unit Value
## 1 UNPD_World Population Prospects_2006 (International estimate) Years    42
## 2 UNPD_World Population Prospects_2006 (International estimate) Years    42
## 3 UNPD_World Population Prospects_2006 (International estimate) Years    42
## 4 UNPD_World Population Prospects_2006 (International estimate) Years    41
## 5 UNPD_World Population Prospects_2006 (International estimate) Years    42
## 6 UNPD_World Population Prospects_2006 (International estimate) Years    42
##   Value.Footnotes
## 1              NA
## 2              NA
## 3              NA
## 4              NA
## 5              NA
## 6              NA
```

4.15 Life expectancy of men vs. women by country

Let's manipulate the data to make our exploration easier. We will build the dataset for our first plot in which we will represent the average life expectancy

of men and women across countries for the last period recorded in our data (2000-2005).

Task: manipulate the dataset to contain male and female life expectancy for each country.

- Filter `life_expectancy` to obtain all records such as Year is equal to `"2000-2005"`.
- Subset the dataset to include just three columns: `Country.or.Area`, `Subgroup`, and `Value`.
- Have a look at how `pivot_wider()` from the tidyr package is used for converting `Subgroup` into two other columns called `Female` and `Male`, reshaping dataset from long to wide. Try to understand what it's doing. There's nothing to write. Maybe printing the number for rows and columns before and after pivoting will help seeing how the data frame is transformed.
- Print the first rows of the resulting dataset.

```
# Subsetting and reshaping the life expectancy data
subdata <- life_expectancy  %>%
    filter(...) %>%
    select(...) %>%
    pivot_wider(names_from = Subgroup, values_from = Value)

# Taking a look at the first few rows
...
```

```
## # A tibble: 6 x 3
##   Country.or.Area Female  Male
##   <chr>            <int> <int>
## 1 Afghanistan         42    42
## 2 Albania             79    73
## 3 Algeria             72    70
## 4 Angola              43    39
## 5 Argentina           78    71
## 6 Armenia             75    68
```

4.16 Visualize (I)

A scatter plot is a useful way to visualize the relationship between two variables. It is a simple plot in which points are arranged on two axes, each of which represents one of those variables.

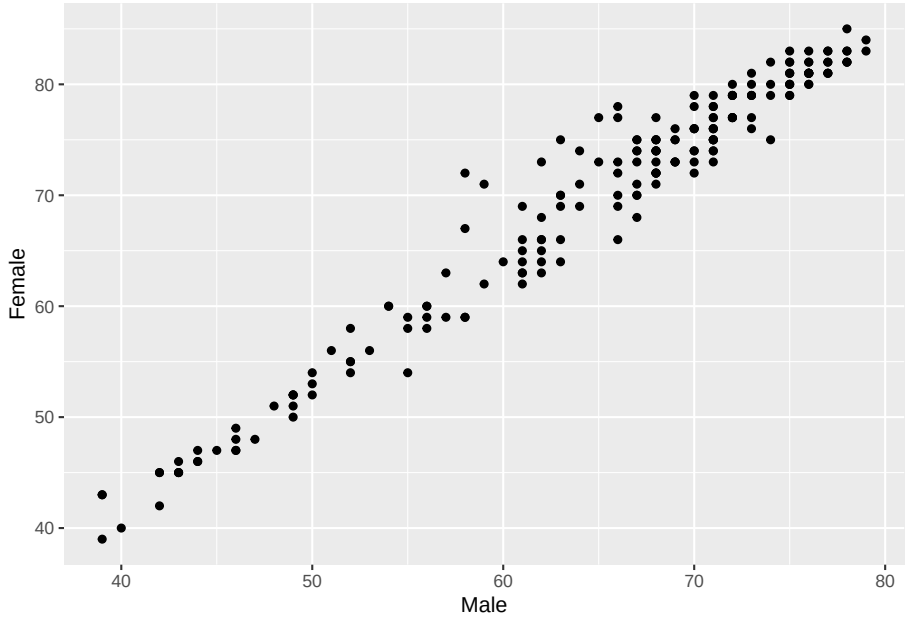
Let's create a scatter plot using ggplot2 to represent life expectancy of males (on the x-axis) against females (on the y-axis). We will create a straightforward

plot in this task, without many details. We will take care of these kinds of things shortly.

For creating a basic scatter plot for male vs. female life expectancy:

- Use the `ggplot()` function to initialize a ggplot object. Declare `subdata` as the input data frame and set the aesthetics to represent `Male` on the x-axis and `Female` on the y-axis.
- Add a layer to represent observations with points using `geom_point()`.

```
# Plotting male and female life expectancy
ggplot(..., aes(...)) +
  ...()
```



4.17 Reference lines (I)

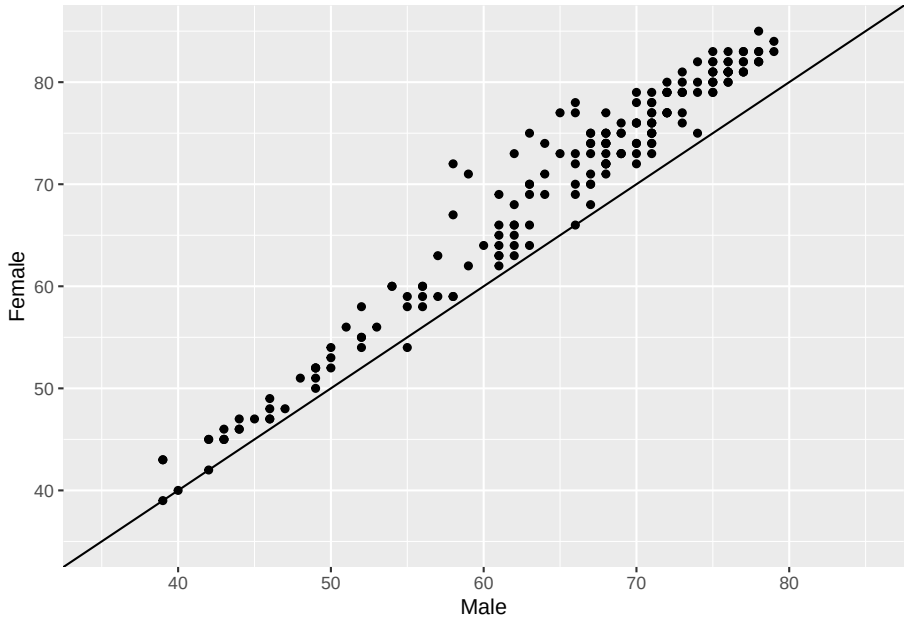
A good plot must be easy to understand. There are many tools in ggplot2 to achieve this goal and we will explore some of them now. Starting from the previous plot, let's set the same limits for both axes as well as place a diagonal line for reference. After doing this, the difference between men and women across countries will be easier to interpret.

After completing this task, we will see how most of the points are arranged above the diagonal and how there is a significant dispersion among them. **What does this all mean?**

Tasks:

- Copy your previous scatter plot code.
- Add a dashed diagonal line that passes by (0, 0) with slope equal to 1.
- Set limit of x-axis from 35 to 85.
- Set limit of y-axis from 35 to 85.

```
ggplot(..., aes(...)) +
  ...() +
  geom_abline(slope = 1, intercept = 0) +
  scale_x_continuous(limits = c(..., ...))+
  scale_y_continuous(limits = c(..., ...))
```



4.18 Plot titles and axis labels

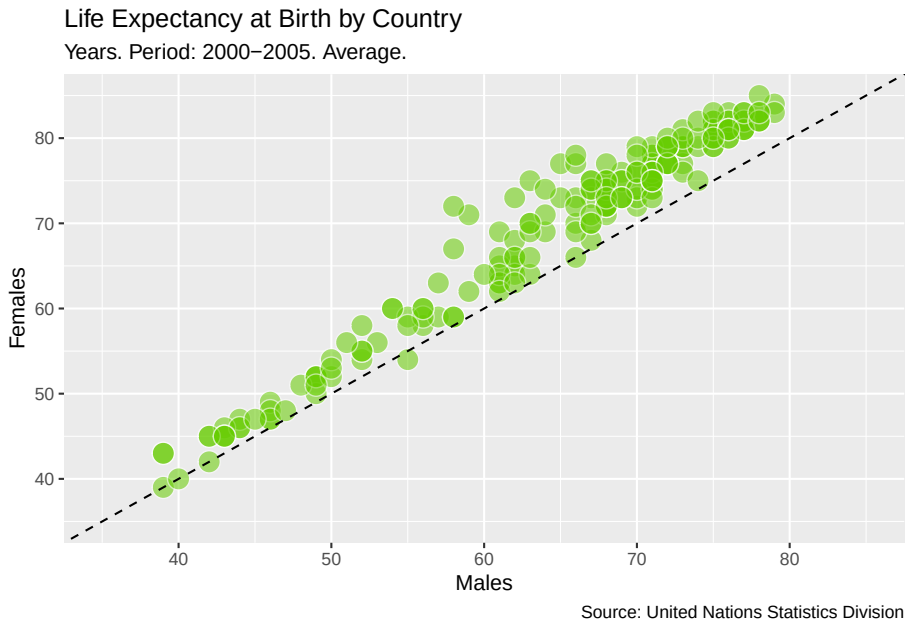
A key point to make a plot understandable is placing clear labels on it. Let's add titles, axis labels, and a caption to refer to the source of data. Let's also change the appearance to make it clearer.

Tasks for adding plot titles and axis labels.

- Add the plot title: `"Life Expectancy at Birth by Country"`.

- Add the next caption: `"Source: United Nations Statistics Division"`.
- Set the x-axis label to `"Males"`.
- Set the y-axis label to `"Females"`.
- Have a look at the changes that can be made to the points, in order to improve the plot a bit (that's done inside the geom_point() call). Make sure you understand the parameters.
- Note that the points can have color and filling if we give them `shape = 21`, which means than it is not just a point, but a circle or bubble, with border and area. This is useful for plotting several variables at the same time with the `aes()` function (one would be mapped to `colour=` and the other to `fill=`).

```
# Adding labels to previous plot
ggplot(..., aes(...)) +
  ...(colour="white", fill="chartreuse3", shape=21, alpha=.55, size=5) +
  geom_...(slope = 1, intercept = 0) +
  scale_x_continuous(limits = c(..., ...)) +
  scale_y_continuous(limits = c(..., ...)) +
  labs(title = ...,
       subtitle = ...,
       caption = ...,
       x = ...,
       y = ...)
```



4.19 Highlighting remarkable countries (I)

Now, we will label some points of our plot with the name of its corresponding country. We want to draw attention to some special countries where the gap in life expectancy between men and women is significantly high. These will be the final touches on this first plot.

Tasks:

- Modify the `ggplot(...)` function to set the label parameter to Country.or.Area.
- Add a label to countries defined by `top_male` (the data frame is built for you). Note that this consists on adding a new layer to the plot.
- Add a label to countries defined by `top_female` (the data frame is built for you). Note that this consists on adding a new layer to the plot.
- Change the plot theme to `theme_bw()`. Note that this consists on adding a new layer to the plot. The plot will be lighter, with less elements.
- Make sure you copy and paste anything missing from the previous plot.

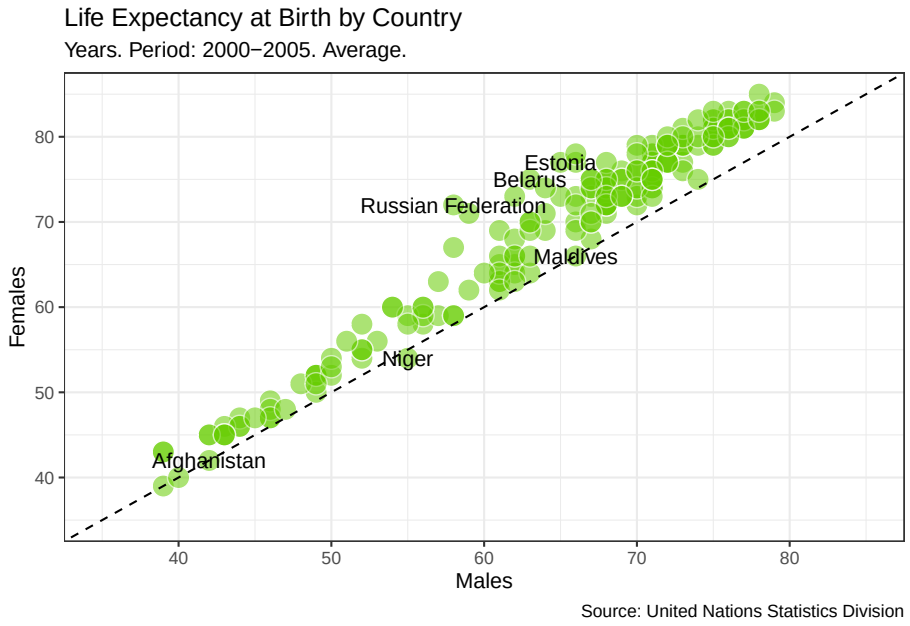
```
# Subseting data to obtain countries of interest
top_male <- subdata %>%
  arrange(Male-Female) %>%
  head(3)

top_female <- subdata %>%
  arrange(Female-Male) %>%
  head(3)

top_male
```

```
## # A tibble: 3 x 3
##   Country.or.Area    Female  Male
##   <chr>               <int> <int>
## 1 Russian Federation     72    58
## 2 Belarus                75    63
## 3 Estonia                77    65
```

```
top_female
```

```
## # A tibble: 3 x 3
##   Country.or.Area Female  Male
##   <chr>            <int> <int>
## 1 Niger               54    55
## 2 Afghanistan         42    42
## 3 Maldives            66    66
```

```
# Adding text to the previous plot to label countries of interest
ggplot(..., aes(...)) +
  ...(... = "white", ... = "chartreuse3", shape = 21, alpha = .55, size = 5) +
  geom_...(slope = 1, intercept = 0) +
  geom_text(data = top_male) +
  geom_...(data = ...) +
  scale_x_continuous(limits = c(..., ...)) +
  scale_y_continuous(limits = c(..., ...)) +
  labs(title = ...,
       subtitle = ...,
       caption = ...,
       x = ...,
       y = ...) +
  theme_bw()
```



4.20 GIF

We can also include information about the evolution over time of these data. There are several periods of five years, so we are going to generate a plot per period, so that we have like a movie of plots.

The easy way of doing this is using the gganimate package, but it only works with numerical data or dates. Here we have a character for the `Year` column,

with data in this format: `"1990-1995"`. We will trick this a bit so that we can take advantage from the library. We transform the data frame with dplyr and tidyr.

Remember that before using the data frame for the plot, we have to transform it into a wide format. Again, we do this with tidyr and its `pivot_wider()` function. Then, we use `mutate()` for changing the `Year` column. We use `stringr` for getting the first year of the period (originally, the column is a factor but we can work with it as if it were a character). Then we change it to numeric. This code has been provided for you.

Finally, we crate a data frame just with the Spanish data. We will use this for tracking Spain during all the evolution.

```
df_subdata2 <- life_expectancy %>%
  select(Country.or.Area, Subgroup, Year, Value) %>%
  filter(Subgroup %in% c("Female", "Male")) %>%
  pivot_wider(names_from = ..., values_from = ...) %>%
  mutate(Year = stringr::str_sub(Year, 1, 4),
         Year = as.numeric(Year))

df_country_selection <- df_subdata2 %>%
  filter(...)
```

Now the data frame has been created, we use it as input for the plot.

- First, we call the library gganimate. It includes a function `transition_time()`, which is used a new layer in the plot. It receives the column of the data frame that will be used as time. In this function, even though we are using a column from the data frame, **we don't use the `aes()` function.**
- We replicate the previous plot. The only change is that we are only using one `geom_text()` layer, for the Spanish data. It should receive the `df_country_selection` data frame in the `data=` parameter.
- Have a look a the `subtitle=` label. Instead of writing the same as before, we now want it to vary depending on the year. The `transition_time()` creates a variable called `frame_time` that will indicate which `Year` is using for each frame. The original format it generates is ugly, so we adapt it with the `scales::number()` function. The curly brackets `{` are important in order to indicate R that what is within them must be evaluated. If we didn't use them, R would understand that operation as a simple character. For a better understanding on how the curly bracktes work, try not writing them.
- The `transition_time()` receives the column used for the time evolution, `Time` in this case. Remember, without `aes()` function.

```
library(...)

ggplot(...)+
  geom_...(...) + # points
  geom_...(...) + # line
  geom_...(data = ...) + # text
  scale_x_...(...) +
  scale_y_...(...) +
  labs(title="Life Expectancy at Birth by Country",
       subtitle="Years. Period: {scales::number(frame_time)}. Average.",
       caption="Source: United Nations Statistics Division",
       x="Males",
       y="Females")+
  theme_bw() +
  transition_time(...)
```

Chapter 5

Programming concepts

5.1 Purpose of functions

Simulating random data consists on generating a sample of numbers that seems random (this sounds vague but there are statistical ways of quantifying this similarity). For doing this, you need an algorithm that from a seed (a beginning number) will make several calculations that will return a new number. This new number is used for calculating the next. You can force that these generated numbers will have a gaussian distribution, or uniform, or be binary (as if they were heads and tails), whatever you need.

You may know some algorithms that perform this, but coding it is tedious. In R (or any statistical tool) you can easily generate such a sequence with just a tiny piece of code. You just need a function.

Imagine you want to randomly simulate some numbers with a gaussian distribution. You could study the algorithm and coding it, and execute that code whenever you need a new simulation, or you can use the `rnorm()` function.

A function is a piece of code with a name (*rnorm* in this case) that will execute the code that it represents (behind `rnorm()` that generator algorithm is written).

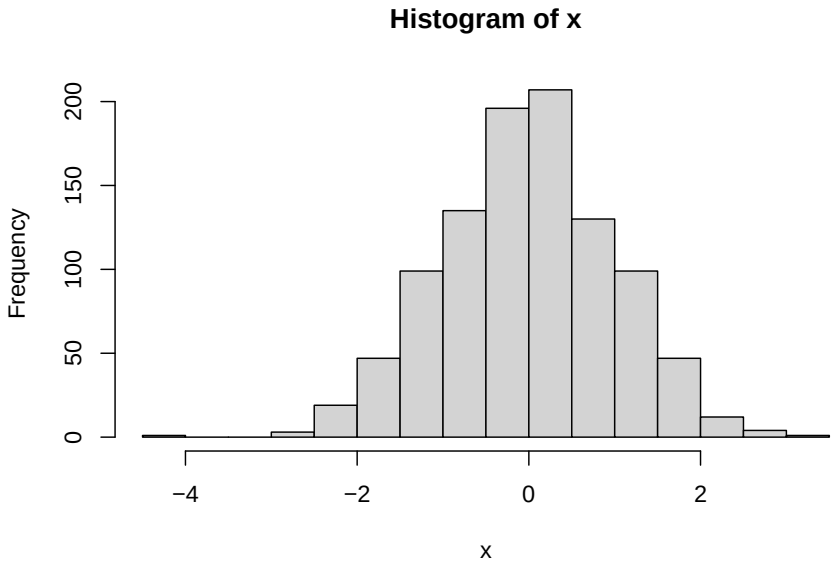
```
x <- rnorm(1000)
head(x)
```

```
## [1] -1.5987095  1.0220178 -0.5928370 -0.5387604 -1.6497944  0.2080619
```

`head()` is also a function. You already know it. It shows the first six elements of what it receives.

We can now have a look at the entire generated vector to confirm it is gaussian, maybe with a plot. For intance, a histogram

```
hist(x)
```



The `hist()` function receives a vector and plots its histogram. Can you imagine having to plot manually a histogram like this every time a new vector is generated? Fortunately, with just one word you can get it done.

5.2 The arguments

We are going to work with randomly generated data. This is a bit tricky. A computer needs always a formula to do something, therefore nothing is random. Random data seems random according to some statistics but it is deterministic. This means that you can always replicate the generation of the *random* data. For doing this, take into account that this generation begins from a number, called seed. If you use the same seed I use, the data should be the same.

This is how to set it up.

```
set.seed(31818)
```

So now we generate a vector of random numbers. They have a normal distribution (we will not get into probability theory during this course).

```
algunos_numeros <- rnorm(10)

algunos_numeros
```

```
##  [1] -0.3584224 -0.3227992  0.7659481 -1.5462213  0.3149461  0.6487038
##  [7] -0.9278019 -1.7762911  0.6025721  0.3918201
```

These are just numbers, so we can calculate something like the mean. For doing this we use the function `mean()`.

```
mean(algunos_numeros)
```

```
## [1] -0.2207546
```

But what is a function? It is a piece of code with a name that do something with generated objects. It can generate objects when you call it our use already generated ones. For instance, what we have just done is calculating the mean of a set of numbers we generated a few lines above.

This function receives a set of numbers and computes their mean but it can do something else. Imagine some of these numbers are `NA`. We should know by now that the mean of `NA` is `NA`, thus if we have some `NA` data within our numbers, we'll get `NA` when computing the mean (and other calculations).

```
algunos_numeros[c(3, 6)] <- NA
algunos_numeros
```

```
##  [1] -0.3584224 -0.3227992         NA -1.5462213  0.3149461         NA
##  [7] -0.9278019 -1.7762911  0.6025721  0.3918201
```

```
mean(algunos_numeros)
```

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

For avoiding this results, we pass into the function more information. We can say the function not to take into account `NA` data. Doing this, the function we'll do as desired. The way of doing this is using a parameter or argument.

We can use the `na.rm=` argument to deal with the `NA` values.

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

```
## [1] -0.4527747
```

We could try to calculate the mean of `algunos_numeros` manually:

```
sum(algunos_numeros) / length(algunos_numeros)
```

It requires more coding. And we should include the `na.rm=TRUE` argument in the `sum()` function and take this into account for recalculating the length of the vector, since the `length()` function doesn't have any argument for doing this. The `mean()` simplyfies all this.

Something very important is the fact that `algunos_numeros` in our example was also another argument for `mean()`. This function has several arguments and the first one is the set of numbers whose mean we want to calculate. Let's get into this.

5.2.1 Order of the arguments

```
mas_numeros <- runif(23)
mas_numeros
```

```
##  [1] 0.53859382 0.07238508 0.78758628 0.64420567 0.54027124 0.31568365
##  [7] 0.49614804 0.86409239 0.80507278 0.47002802 0.39032079 0.04387482
## [13] 0.58063723 0.22943255 0.42943311 0.74572542 0.34477960 0.41626907
## [19] 0.84390398 0.76753972 0.56551513 0.43716761 0.31449825
```

The `runif()` function has also several arguments. We can have a look at the documentation and see that it takes three of them:

- `n`, the number of observations that will be randomly generated;
- `min`, the minimum number allowed on the simulation;
- `max`, the maximum number allowed on the simulation.

When we wrote `runif(23)` we were specifying the `n=` argument. Reading the documentation we can see that there is no need on doing this for `min=` or `max=` since values have been set for them by default. For changing them, we write them explicity.

```
mas_numeros <- runif(23, min = -15, max = -1)
mas_numeros
```

```
##  [1]  -4.019444  -3.399739  -2.931166  -2.672629 -13.986012  -9.916109
##  [7] -14.908869  -6.222678  -2.583648  -1.779950  -5.043804  -8.503062
## [13] -12.329554  -3.960020  -2.801435  -1.528991  -9.156267 -10.872894
## [19]  -2.487192 -13.946144  -3.788698  -8.698802  -7.250312
```

Note that `runif(23, -15, -1)` or `runif(n = 23, min = -15, max = -1)` work as well (apart from the fact that it is randomly generated data and it changes with every execution). If you write the name of the argument, everything works perfectly. If you don't write the name, you must make sure that the arguments are in the same orden as shown in the documentation. Read **always** the documentation in order to understand the order. In case there are many arguments in the function, you should write them all, even if they are in the correct order: it'll make the reading easier.

5.3 The value

Almost always functions will return a result. Functions receive (usually) something as an argument. This *something* is an object, like a number, a vector, a data frame... It works with it, transforming it or making some calculations based on it and return a final result, that may be another number, data frame, a plot, a complex object,... Anything.

Remark. It is not mandatory returning something with a function. We may see some examples later on.

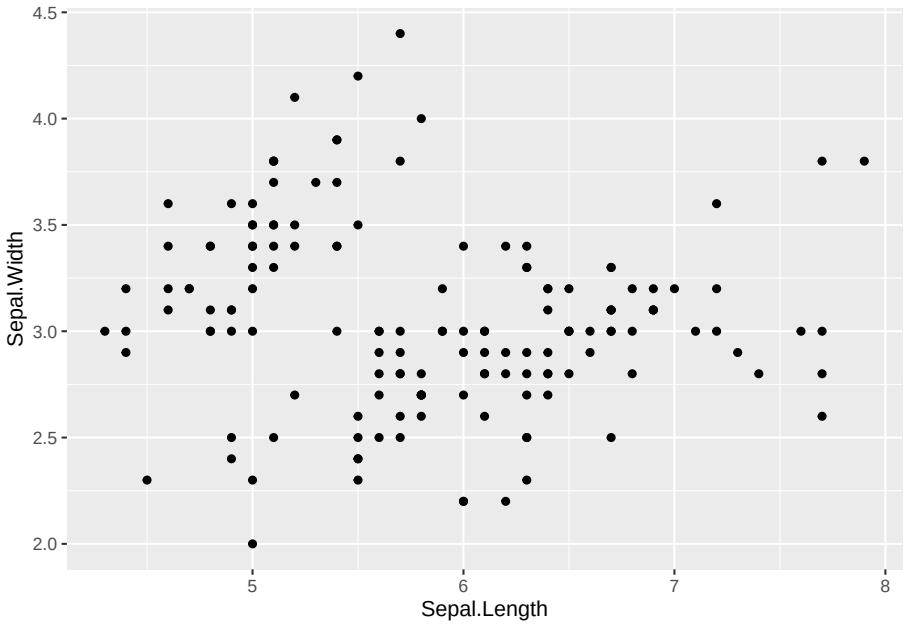
The `mean()` function returns a number, but the `ggplot()` function returns a plot.

```
library(ggplot2)
P <- ggplot(iris)
class(P)
```

```
## [1] "gg"     "ggplot"
```

It is important understanding what can be retrieved from a function in order to proceed working with that result. For example, since `P` is a `ggplot` object we know that we can add `ggplot` layers to it, and these layers can be generated in fact with more function.

```
P + geom_point(aes(x = Sepal.Length, y = Sepal.Width))
```



Everything you do in R is a function call. Remember that: everything. Therefore, the more functions you know, the more you can do.

What happens when there is no function useful for what you want do? You create new one.

5.4 Your own functions

We have during the whole course using functions: `mean()`, `filter()`, `ggplot()`,... All those **pieces of code with parenthesis were functions.** Every action we do in R is a call to a function.

Sometimes, we need functions that have not been created yet. Bu we can create them. We do this with another function: `function()`.

For creating one, we need a name. Let's create one called `mi_media`. It receives a set of numbers and will calculate its mean using `mean()`. Before this calculation, it will print a message.

Let's see the skeleton:

```
mi_media <- function(numeros_que_recibo){

  print("Calculando...")
  mean(numeros_que_recibo)
```

```
}
```

Our function is very simple but has all what every function should have:

- **The name.** We will use this for calling it, in the same you call a friend by his or her name.
- **Arguments.** If the function receives a set of numbers of doing something with them, we need something for representing them. We don't know yet what numbers we will use, but we can represent them in an abstract way. We do this with arguments that will allow us parametrizing these numbers. It works as an x in an equation. We don't know the value of x but we can operate with it. For instance, if $x - 4 = 1$, we can also say that $x - 1 = 4$. We don't know the value of x but we can work with it. In our function, we don't know the value of `numeros_que_recibo` but we know that it should be a vector with numbers, so we can use it as if it were that. So we can calculate its mean.
- **Body.** It is what our function does. The arguments here will be used in an abstract way but later on we will call the function and they'll get real values. Then, the function will work with these real values and we'll get a result.

```
mi_media(mas_numeros)
```

```
## [1] "Calculando..."
```

```
## [1] -6.642931
```

We can work with several arguments in our own function.

Let's create a new function and see what happens when we change the order of the arguments.

```
mi_potencia <- function(x, y) {
# function to print x raised to the power y
  resultado <- x^y
  print(paste(x,"elevado a", y, "es", resultado))
}
```

```
mi_potencia(8, 2)
```

```
## [1] "8 elevado a 2 es 64"
```

```
mi_potencia(2, 8)
```

```
## [1] "2 elevado a 8 es 256"
```

The first thing the function receives is the `x` and then the `y`. If we don't say anything, this is always like this for this `mi_potencia` function. This is why is not the same using the 2 first and the 8 secondly.

```
mi_potencia(8, 2)
```

```
## [1] "8 elevado a 2 es 64"
```

```
mi_potencia(x = 8, y = 2)
```

```
## [1] "8 elevado a 2 es 64"
```

```
mi_potencia(y = 8, x = 2)
```

```
## [1] "2 elevado a 8 es 256"
```

But we can say explicitily what the `x` and the `y` is so the function we'll not make assumptions.

Now let's make, by default, the exponent equals to 2.

```
mi_potencia <- function(x, y = 2) {
  resultado <- x^y
  print(paste(x,"elevado a", y, "es", resultado))
}
```

```
mi_potencia(10)
```

```
## [1] "10 elevado a 2 es 100"
```

What we have done is fixing a value for `y`, so there is no need to specify the value. It is 2 by default. If we want to change it, we just mention it when we call the function.

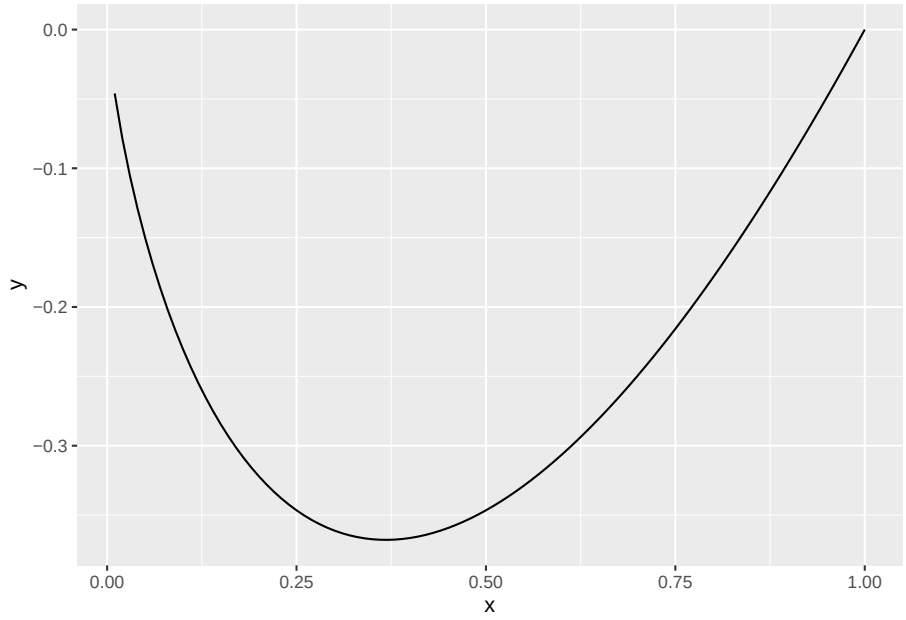
```
mi_potencia(10, 3)
```

```
## [1] "10 elevado a 3 es 1000"
```

5.5 Exercises

1. Create a function that receives a vector `x` of numbers and a number `a` and plots, based on them, the curve of the mathematical function $f(x) = xln(ax)$ being a a positive real number. Make sure that `a` equals 1 by default. The function should include this vector as the column of a data frame (`tibble()` function is needed). Then it will create another column with the formula of the mathematical function, with dplyr. Finally, with ggplot2, it will plot the curve. A line plot will suffice. For checking that the function works correctly, call it with the vector generated by this code: `seq(0.01, 1, by = 0.01)`. Try different `x` and `a` cases to see how the plot changes.

```
library(dplyr)
library(ggplot2)

xlnx <- function(x, a = 1){

  df <- tibble(x = x) %>%
    mutate(y = x * log(a * x))

  ggplot(df) +
    geom_line(aes(x = x, y = y))
}

xlnx(seq(0.01, 1, by = 0.01))
```




```
# xlnx(seq(1, 100, by = 1), a = 0.01)
```

2. In the function you created before, include a piece of code that will avoid accepting cases where the logarithm will be evaluated under negative numbers.

3. We are going to define a function that reads a csv about air quality in Madrid, fix the format of two columns and returns it. After that, we will merge all the information from the files into one only data frame. The files are the ones that begin with `NO2` or `CO`. First, the function will receive the path of a file. It reads it (do this with readr; guess the function) and now we have to use dplyr for fixing the columns `mes` and `dia`. We want them to be numeric. `mes` is easy but for `dia` you will need to use some character treatment. Try to use `str_remove()` from the stringr package. Now make sure the function returns the data frame you created. Also, exclude the `magnitud` function: it is not necessary because the last columns specifies the magnitude. The joining stuff is the next exercise. For now, make sure that the function works for the six data frames.

```
library(stringr)

trata_fichero <- function(ruta_al_fichero){
  df <- read_csv(ruta_al_fichero) %>%
    mutate(mes = as.numeric(mes),
```

```
            dia = str_remove(dia, "d"),
            dia = as.numeric(dia),
            magnitud = NULL)

  return(df)
}

# trata_fichero("../../data/NO2_2017.csv")
# trata_fichero("../../data/NO2_2018.csv")
trata_fichero("data/NO2_2019.csv")
trata_fichero("data/CO_2019.csv")
```

4. Now we want to mix all the information. Create one data frame with all the data about NO2 and another one about CO. Use `bind_rows()` from dplyr for this. After that, join them all with `inner_join()`. How many rows are there? Do the same with `full_join()`, `right_join()`, and `left_join()`. What differences do you see?

```
inner_join(
  bind_rows(
    trata_fichero("data/NO2_2017.csv"),
    trata_fichero("data/NO2_2018.csv"),
    trata_fichero("data/NO2_2019.csv")
  ),
  bind_rows(
    trata_fichero("data/CO_2017.csv"),
    trata_fichero("data/CO_2018.csv"),
    trata_fichero("data/CO_2019.csv")
  )
) %>%
  nrow()
```

5. Could you guess what the result of the next code is, without executing it? What is the rol of each `c`?

```
c <- 1
c(c = c)
```

6. The `letters` vector contains all the letter from `"a"` to `"z"`. You can select a random sample with the `sample()` function. Create a function that receives an integer number n, then it creates a sample of n letters. After that, it sorts the letters and finally it collapses them all with the `paste0()` function and the `collapse=` attribute.

7. Create a function similar to the previous one but, instead of generating a sample from `letters`, it generates two samples: one with `letters` and one from `LETTERS`. The function must receive one argument for the number of elements from `letters` and another one for `LETTERS`. Then it will create one vector with the two samples (use `c()`). Now sort that vector and collapse all the letters.

8. Create a function that receives two columns from the `iris` data frame as two characters and multiplies these two columns. Create another one with the this product you just calculated. Do **not** use dplyr.

9. Create a function that reads the data frame about Human Resources data. The path should be one of the arguments of the function. Then it will select all the columns except sales and salary. Now generate a random logical vector whose length will be the number of rows of the data frame. This vector should have a bigger amount of `TRUE`s than `FALSE`s: the weight should be an attribute of the function (it consists just on a vector of two numbers whose addition is 1, e.g., 0.7 and 0.3). This will be used for splitting the data frame into two data frames. This can be done with `filter()`. Then use the `glm()` for developing a predictive model with the first part of the data frame (the biggest one). Then use the other data frame for making a prediction with the `predict()` model. You can check how many cases you have correctly predicted with the `table()` function. Make the function returning this table. You have just developped a model for predicting which employees are more inclined to leaving the company.

10. Have a look at these codes and try to guess which will be the value of `a`, `aa` and `aaa`. Don't run the code until you have an idea of the final result.

```
a <- 1

mi_funcion <- function(b){
  if(b > 0){
    a <- 100
  } else {
    a <- -50
  }

}

mi_funcion(10)
a
```

```
aa <- 0

mi_funcion2 <- function(b){
```

```
  if(is.character(b)){
    aa <- aa + 10
  } else if(is.numeric(b)){
    aa <- aa - 20
  }

}

mi_funcion2("hola")
aa
```

```
mi_funcion3 <- function(aaa, b){
  aaa <- aaa + b
  return(aaa)
}

b <- mi_funcion3(3, 4)
aaa
```

11. Create a function with two attributes. The first one is called `case=` and it will receive a word. The second one is called `times=` and it receives an integer number. If the word it receives is `"dados"` or `"Dados"`, it generates two vectors of length `times`, with values from 1 to 6. Create another vector adding these two vectors and calculate which is the more frequent result. If the word is `"coin"` or `"Coin"`, then simulate flipping a coin with the `bernoulli()` function and calculate the probability of getting each result. Flip the coin the number of times indicated in the `times=` attribute of the function. If the received word is something else, print `"I don't know what to do with this"`.

5.6 Conditions

```
a <- 5
if(a > 2){
  print("a is greater than 2")
} else {
  print("a is lower or equal to 2")
}
```

```
## [1] "a is greater than 2"
```

```
b <- 10
if(b > 0 & b < 8){
  print("I like b")
}
```

```
a <- 5
b <- 2

if(a >= b){
  c <- "greater"
} else {
  d <- "lower"
}
```

```
a <- 10
b <- -3

if(a %% 2 == 0 & b < 0){
  c <- a / 2 - b
} else {
  c <- a
}
c
```

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

```
a <- 10
b <- -3

if(a %% 2 == 0 & b %% 2 == 0){
  c <- a / 2 - b / 2
} else if(a %% 2 & b < 0){
  c <- a / 2 - b
} else {
  c <- a + b
}
c
```

```
## [1] 7
```

5.7 Joining data frames with dplyr

```
library(readr)
library(dplyr)

df_hombres <- read_tsv("data/evolutivo_poblacion_varones.datos", col_types = cols())
df_mujeres <- read_tsv("data/evolutivo_poblacion_mujeres.datos", col_types = cols())

glimpse(df_hombres)
```

```
## Rows: 750
## Columns: 3
## $ provincia <chr> "02 Albacete", "02 Albacete", "02 Albacete", "02 Albacete", ~
## $ year      <dbl> 2018, 2017, 2016, 2009, 2008, 2007, 2006, 2005, 2004, 2003, ~
## $ hombres   <dbl> 194628, 195289, 196277, 199729, 197673, 195338, 193310, 1920~
```

```
glimpse(df_mujeres)
```

```
## Rows: 816
## Columns: 3
## $ provincia <chr> "02 Albacete", "02 Albacete", "02 Albacete", "02 Albacete", ~
## $ year      <dbl> 2018, 2017, 2015, 2013, 2012, 2011, 2010, 2009, 2008, 2007, ~
## $ mujeres   <dbl> 194158, 194743, 197014, 200016, 201640, 201400, 201396, 2011~
```

```
df_hombres_2017 <- df_hombres %>%
  filter(year == 2017)

df_mujeres_2017 <- df_mujeres %>%
  filter(year == 2017)

nrow(df_mujeres_2017)
```

```
## [1] 51
```

```
nrow(df_hombres_2017)
```

```
## [1] 50
```

Hay más filas en un data frame que en otro: estas tablas han sido tratadas previamente y en uno caso falta una provincia y en otro, dos.

```
df_union_2017 <- inner_join(df_mujeres_2017, df_hombres_2017)
```

```
## Joining, by = c("provincia", "year")
```

```
glimpse(df_union_2017)
```

```
## Rows: 49
## Columns: 4
## $ provincia <chr> "02 Albacete", "03 Alicante/Alacant", "04 Almeria", "01 Arab~
## $ year      <dbl> 2017, 2017, 2017, 2017, 2017, 2017, 2017, 2017, 2017, 2017, ~
## $ mujeres   <dbl> 194743, 904711, 359676, 161354, 493911, 80674, 336566, 56180~
## $ hombres   <dbl> 195289, 920621, 346996, 165220, 541049, 80026, 343318, 56710~
```

```
df_union_2017_2 <- left_join(df_mujeres_2017, df_hombres_2017)
```

```
## Joining, by = c("provincia", "year")
```

```
nrow(df_union_2017_2)
```

```
## [1] 51
```

```
df_union_2017_2 %>%
  filter(is.na(hombres))
```

```
## # A tibble: 2 x 4
##   provincia    year mujeres hombres
##   <chr>       <dbl>   <dbl>   <dbl>
## 1 18 Granada   2017  449821      NA
## 2 34 Palencia  2017   80943      NA
```

```
df_union_2017_3 <- full_join(df_mujeres_2017, df_hombres_2017)
```

```
## Joining, by = c("provincia", "year")
```

```
nrow(df_union_2017_3)
```

```
## [1] 52
```

```
df_union_2017_3 %>%
  filter(is.na(mujeres))
```

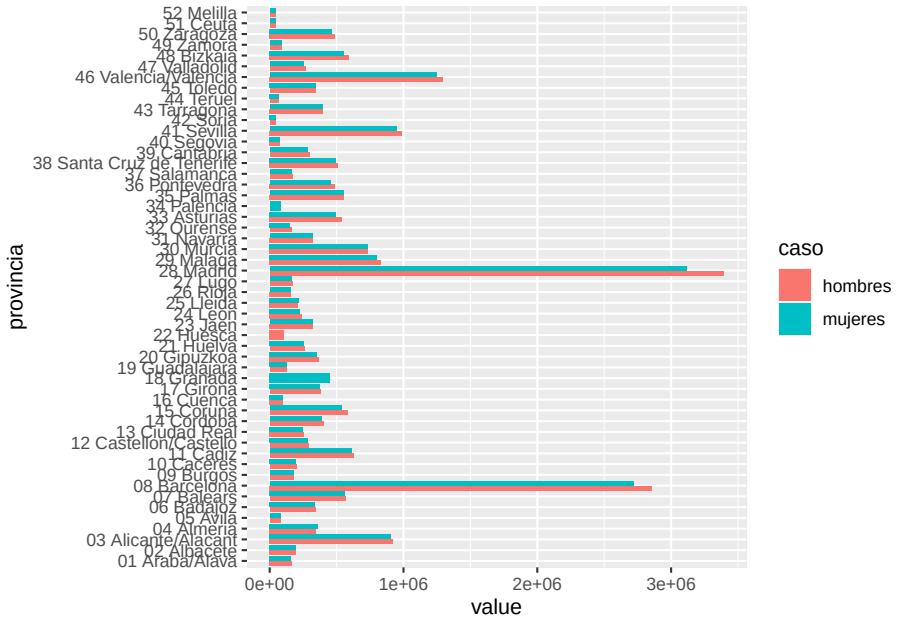
```
## # A tibble: 1 x 4
##   provincia  year mujeres hombres
##   <chr>     <dbl>   <dbl>   <dbl>
## 1 22 Huesca  2017      NA  108848
```

5.8 Binding data frames

```
df_hombres_2017_aux <- df_hombres_2017 %>%
  rename(value = hombres) %>%
  mutate(caso = "hombres")

df_mujeres_2017_aux <- df_mujeres_2017 %>%
  rename(value = mujeres) %>%
  mutate(caso = "mujeres")

df_2017 <- bind_rows(df_hombres_2017_aux, df_mujeres_2017_aux)

glimpse(df_2017)
```

```
## Rows: 101
## Columns: 4
## $ provincia <chr> "02 Albacete", "03 Alicante/Alacant", "04 Almeria", "01 Arab~
## $ year      <dbl> 2017, 2017, 2017, 2017, 2017, 2017, 2017, 2017, 2017, 2017, ~
## $ value     <dbl> 195289, 920621, 346996, 165220, 541049, 80026, 343318, 56710~
## $ caso      <chr> "hombres", "hombres", "hombres", "hombres", "hombres", "homb~
```

```
library(ggplot2)
ggplot(df_2017) +
  geom_col(aes(x = provincia, y = value, fill = caso), position = "dodge") +
  coord_flip()
```



5.9 Exercises

1. Create a function that join the `df_hombres` and `df_mujeres` data frames. It will receive a number that will be a year used as a filter. It will also receive a word for deciding among the four cases: if the word is `"left"`, it will join via `left_join()`, if it is `"inner"`, via `inner_join()`, and so on. If the word is not `"left"`, `"inner"`, `"right"` or `"full"` it will print a message saying `"I don't know what to do with this"`.

```
funcion1 <- function(anno, case){

  df_hombres_filtrado <- df_hombres %>%
    filter(year == anno)

  df_mujeres_filtrado <- df_mujeres %>%
    filter(year == anno)

  if(case == "left"){
    df_hombres_filtrado %>% left_join(df_mujeres_filtrado)
  } else if(case == "inner"){
    df_hombres_filtrado %>% inner_join(df_mujeres_filtrado)
  } else if(case == "right"){
    df_hombres_filtrado %>% right_join(df_mujeres_filtrado)
```

```
  } else if(case == "full"){
    df_hombres_filtrado %>% full_join(df_mujeres_filtrado)
  } else {
    print("I don't know what to do with this")
  }

}

funcion1(2017, "inner")
funcion1(2017, "left")
funcion1(2017, "right")
funcion1(2017, "full")
```

2. Add to the previous function a piece of code that will print a message if the year passed as a parameter is not among the available ones.

```
funcion2 <- function(anno, case){

  annos_disponibles <- c(unique(df_hombres$year), unique(df_mujeres$year))

  # With stop() I force an error in case the selected year is not available. If you receive an er
  if(!anno %in% annos_disponibles){
    stop("The selected year is not available")
  }

  df_hombres_filtrado <- df_hombres %>%
    filter(year == anno)

  df_mujeres_filtrado <- df_mujeres %>%
    filter(year == anno)

  if(case == "left"){
    df_hombres_filtrado %>% left_join(df_mujeres_filtrado)
  } else if(case == "inner"){
    df_hombres_filtrado %>% inner_join(df_mujeres_filtrado)
  } else if(case == "right"){
    df_hombres_filtrado %>% right_join(df_mujeres_filtrado)
  } else if(case == "full"){
    df_hombres_filtrado %>% full_join(df_mujeres_filtrado)
  } else {
    print("I don't know what to do with this")
  }

}
```

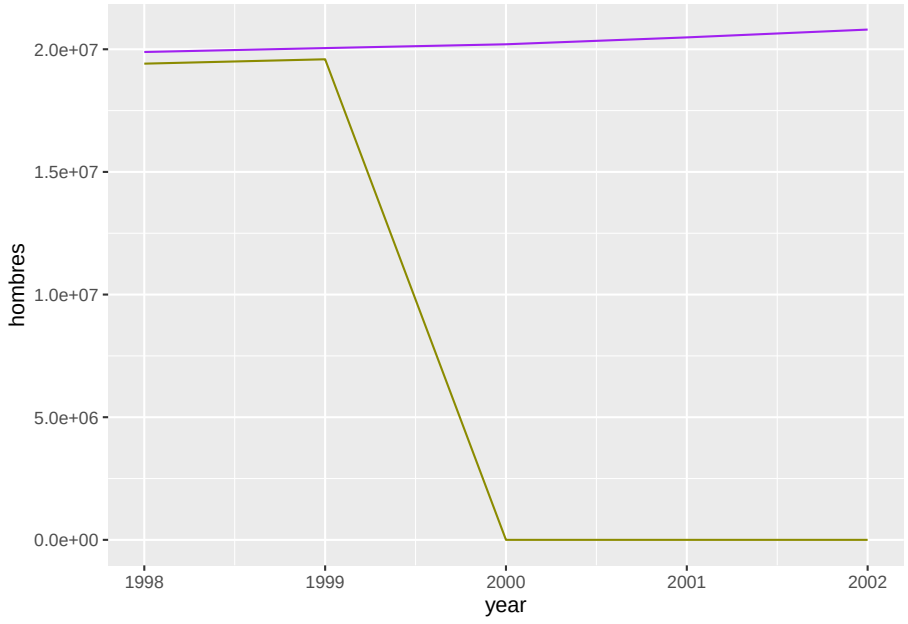
```
funcion2(1980, "inner")
# Error in funcion2(1980, "inner") : The selected year is not available
```

3. Pick the first five years of history and plot the evolution of the men and women population along these years.

```
annos_seleccionados <- c(1998, 1999, 2000, 2001, 2002)

df_hombres_filtrado <- df_hombres %>%
  filter(year %in% annos_seleccionados)

df_mujeres_filtrado <- df_mujeres %>%
  filter(year %in% annos_seleccionados)

df_para_grafico <- df_hombres_filtrado %>%
  full_join(df_mujeres_filtrado)
```

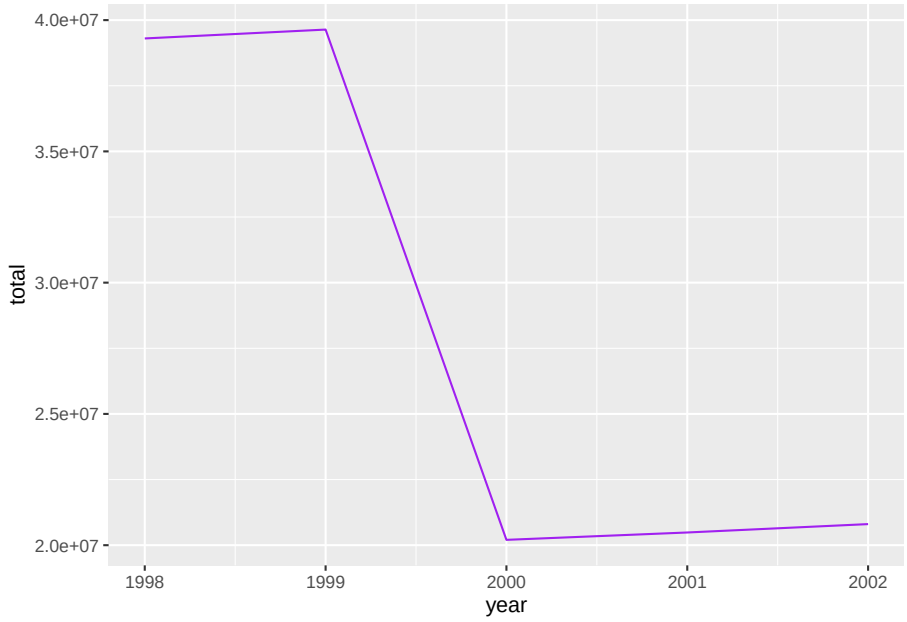
```
## Joining, by = c("provincia", "year")
```

```
df_para_grafico <- df_para_grafico %>%
  group_by(year) %>%
  summarise(
    hombres = sum(hombres, na.rm = TRUE),
    mujeres = sum(mujeres, na.rm = TRUE)
  )


ggplot(df_para_grafico) +
  geom_line(aes(x = year, y = hombres), col = "purple") +
  geom_line(aes(x = year, y = mujeres), col = "yellow4")
```



4. Plot the same evolution but for the entire population.

```
df_para_grafico %>%
  mutate(total = hombres + mujeres) %>%
  ggplot() +
  geom_line(aes(x = year, y = total), col = "purple")
```



5.10 Applying the same function to several elements (I)

```
library(readr)
library(dplyr)
```

```
df_terraces_sample <- read_csv("data/ejemplo1_map.csv", col_types = cols())
```

Which is the class of each column?

```
class(df_terraces_sample$id_terraza)
class(df_terraces_sample$id_local)
class(df_terraces_sample$id_distrito_local)
```

```
library(purrr)
listado_clases <- map(df_terraces_sample, class)

listado_clases[1:5]
```

```
## $id_terraza
```

```
## [1] "numeric"
##
## $id_local
## [1] "numeric"
##
## $id_distrito_local
## [1] "numeric"
##
## $desc_distrito_local
## [1] "character"
##
## $id_barrio_local
## [1] "numeric"
```

```
class(listado_clases)
```

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

5.11 Lists

```
un_vector <- c(1, 5, 8, 3)
un_vector
```

```
## [1] 1 5 8 3
```

```
una_lista <- list(1, 5, 8, 3)
una_lista
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 5
##
## [[3]]
## [1] 8
##
## [[4]]
## [1] 3
```

```
un_vector[1] == una_lista[[1]]
```

```
## [1] TRUE
```

```
otra_lista <- list(c(1, 5, 8, 3))
otra_lista
```

```
## [[1]]
## [1] 1 5 8 3
```

```
length(una_lista)
```

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

```
length(otra_lista)
```

```
## [1] 1
```

```
c(1, "dos", 3, TRUE)
```

```
list(1, "dos", 3, TRUE)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "dos"
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] TRUE
```

A list of data frames

```
lista_dataframes1 <- list(iris, mtcars)

head(lista_dataframes1[[1]])
```

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

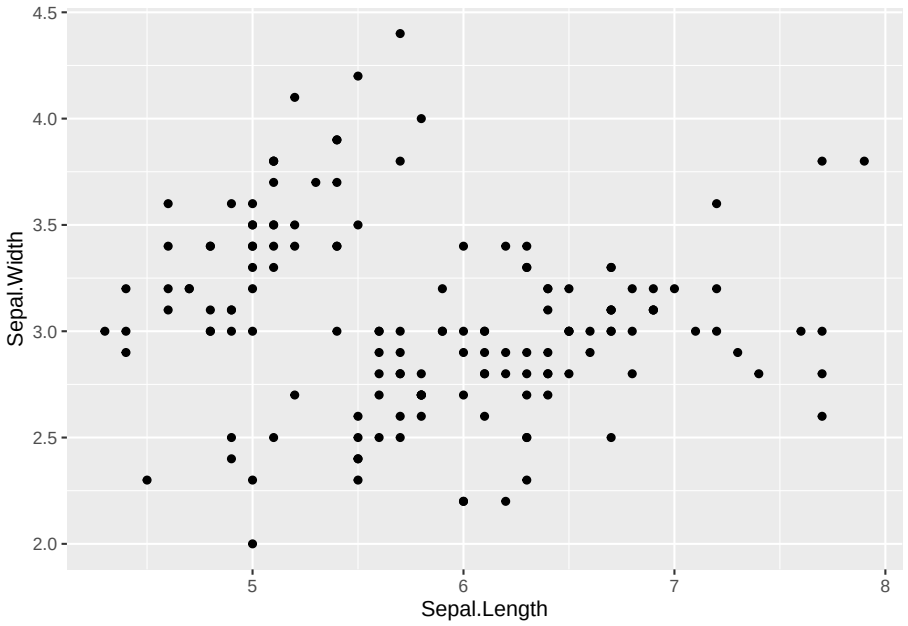
```
head(lista_dataframes1[[2]])
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb cyl_am
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4    6_1
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4    6_1
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1    4_1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1    6_0
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2    8_0
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1    6_0
```

```
library(ggplot2)
P <- ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point()

lista_variada1 <- list(iris, P)

head(lista_variada1[[1]])
```

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

```
lista_variada1[[2]]
```



5.12 Applying the same function to several elements (II)

```
df_co_2017 <- read_csv("data/CO_2017.csv", col_type = cols())
df_co_2018 <- read_csv("data/CO_2018.csv", col_type = cols())
df_co_2019 <- read_csv("data/CO_2019.csv", col_type = cols())
```

They have same structure so we can read them all on the fly and get a list with the three of them.

```
# There are better ways of getting the files' names
files_with_data <- c("data/CO_2017.csv", "data/CO_2018.csv", "data/CO_2019.csv")

files_with_data
```

```
## [1] "data/CO_2017.csv" "data/CO_2018.csv" "data/CO_2019.csv"
```

What we want to do is applying `read_csv()` to each element in that vector. These elements are the files' paths we want to read. So we apply (or map) the `read_csv()` function to these paths.

```
listado_dataframes <- map(files_with_data, read_csv)
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   estacion = col_double(),
##   magnitud = col_double(),
##   ano = col_double(),
##   mes = col_character(),
##   dia = col_character(),
##   dato_co = col_double()
## )
##
##
## -- Column specification --------------------------------------------------------
## cols(
##   estacion = col_double(),
##   magnitud = col_double(),
##   ano = col_double(),
##   mes = col_character(),
##   dia = col_character(),
##   dato_co = col_double()
## )
##
##
## -- Column specification --------------------------------------------------------
## cols(
##   estacion = col_double(),
##   magnitud = col_double(),
##   ano = col_double(),
##   mes = col_character(),
##   dia = col_character(),
##   dato_co = col_double()
## )
```

```
class(listado_dataframes)
```

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

Each element on the list is a data frame, so we can map the common functions we know for data frames to these data frames. Again, with map:

```
map(listado_dataframes, names)
```

```
## [[1]]
## [1] "estacion" "magnitud" "ano"      "mes"      "dia"      "dato_co"
##
## [[2]]
## [1] "estacion" "magnitud" "ano"      "mes"      "dia"      "dato_co"
##
## [[3]]
## [1] "estacion" "magnitud" "ano"      "mes"      "dia"      "dato_co"
```

But we have a list of data frames and we prefer one only data frame gathering all the data. We did this on a previous session with `dplyr::bind_rows()`. So our next goal is *reducing* the list into one data frame. We can do this with `purrr::reduce()`. **Mind the lower case.** With capital *R* is a similar but different function.

```
df_co_unico <- reduce(listado_dataframes, bind_rows)
class(df_co_unico)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

```
map(listado_dataframes, dim)
```

```
## [[1]]
## [1] 3720    6
##
## [[2]]
## [1] 3720    6
##
## [[3]]
## [1] 2790    6
```

```
dim(df_co_unico)
```

```
## [1] 10230     6
```

```
df_unico_directo <- map_df(files_with_data, read_csv)
```

```
##
## -- Column specification --------------------------------------------------------
```

```
## cols(
##   estacion = col_double(),
##   magnitud = col_double(),
##   ano = col_double(),
##   mes = col_character(),
##   dia = col_character(),
##   dato_co = col_double()
## )
##
##
## -- Column specification --------------------------------------------------------
## cols(
##   estacion = col_double(),
##   magnitud = col_double(),
##   ano = col_double(),
##   mes = col_character(),
##   dia = col_character(),
##   dato_co = col_double()
## )
##
##
## -- Column specification --------------------------------------------------------
## cols(
##   estacion = col_double(),
##   magnitud = col_double(),
##   ano = col_double(),
##   mes = col_character(),
##   dia = col_character(),
##   dato_co = col_double()
## )
```

```
dim(df_unico_directo)
```

```
## [1] 10230     6
```

You can also pass parameters to the function you map through the map function.

```
df_unico_directo <- map_df(files_with_data, read_csv, col_types = cols())
head(df_unico_directo)
```

```
## # A tibble: 6 x 6
##   estacion magnitud   ano mes   dia   dato_co
##      <dbl>    <dbl> <dbl> <chr> <chr>   <dbl>
## 1        4        6  2017 01    d01       0.5
```

```
## 2       4       6  2017 01    d02       0.5
## 3       4       6  2017 01    d03       0.7
## 4       4       6  2017 01    d04       0.7
## 5       4       6  2017 01    d05       0.6
## 6       4       6  2017 01    d06       0.6
```

5.13 Exercise

1. In this case `map_df()` works fine but it won't always be that easy. Create a list with `map()` reading `"CO_2017.csv"` and `"NO_2017.csv"`. Now join the two data frames into one only. *Hint.* We already saw how to use the family of functions `*_join()`. Decide which join function you should use and apply with the `reduce()`, as example shown below.
